



.

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

To renew call Telephone Center, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

MAR 17 1992

APR 28 1992

MAR 13 1993

APR 03 1996

JAN 08 1997

FURTHER RESULTS REGARDING MULTIPROCESSOR SYSTEMS

by

Donald Yi-Chung Chang

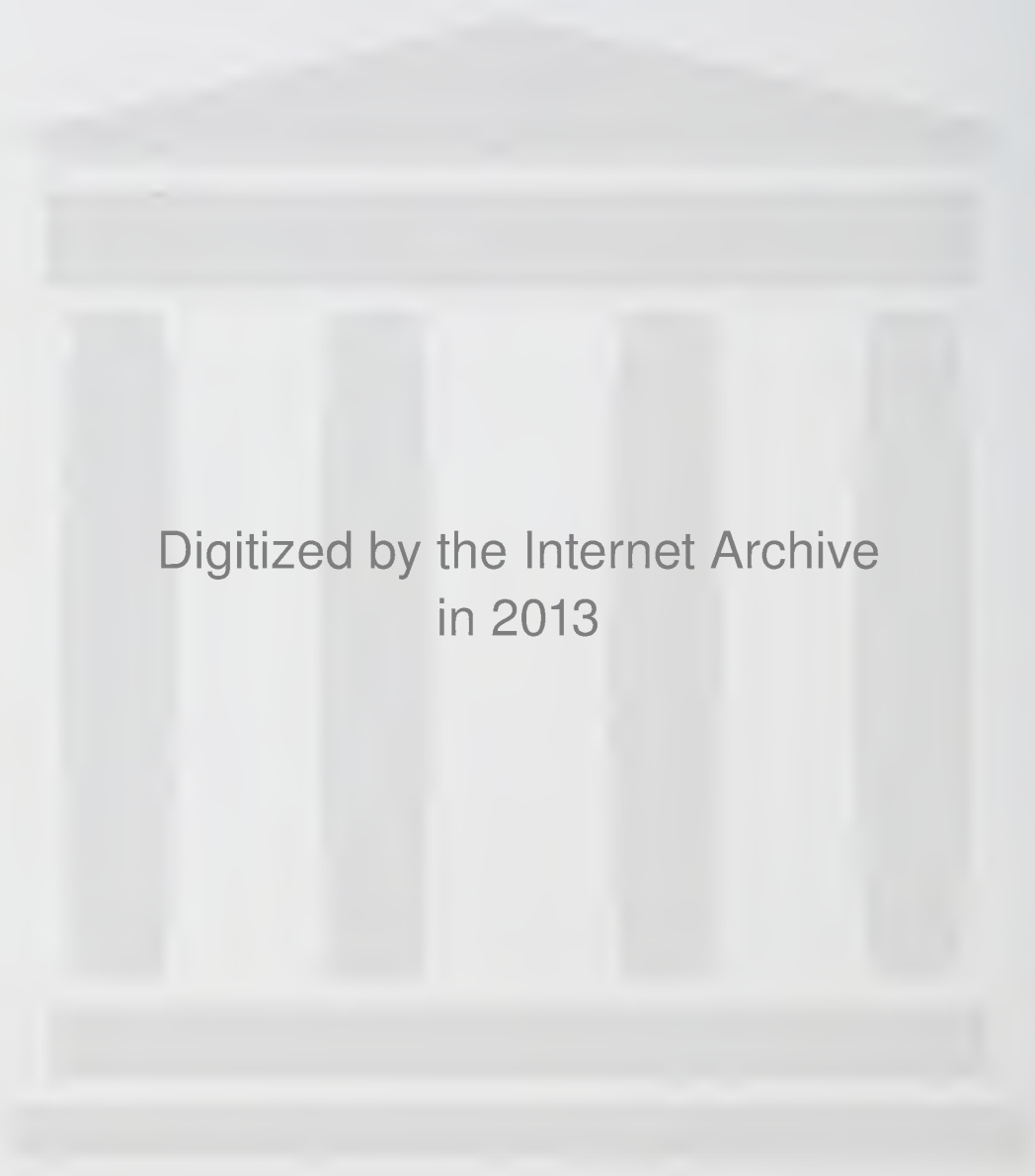
October 1977

NSF-OCA-MCS76-81686-000030



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The Library of the
FEB 5 1978
UNIVERSITY OF ILLINOIS



Digitized by the Internet Archive
in 2013

<http://archive.org/details/furtherresultsre908chan>

Report No. UIUCDCS-R-77-908

FURTHER RESULTS REGARDING MULTIPROCESSOR SYSTEMS^{*}

by

Donald Yi-Chung Chang

October 1977

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

^{*}This work was supported in part by the National Science Foundation under Grant No. US NSF MCS76-81686 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, October 1977.

Acknowledgements

I would like to express my very deep appreciation to my thesis advisor, Professor Duncan H. Lawrie, for his continuous encouragement and guidance throughout this research. It is his pleasant and humorous personality that makes my study at the University of Illinois really enjoyable.

I would also like to thank Professor David J. Kuck for his valuable advice, kind understanding, and most importantly, the long-time financial support.

Thanks are also due to a very dear and special person, Professor C. L. Liu, for his constant moral support and all kinds of help throughout my study here.

Fellow students Wilson K. Wen (now with Sperry-Univac), D. Stott Parker, Jr., Bruce R. Leasure, and Jackson K. C. Hu, who provided valuable comments, encouragement, and friendship are greatly appreciated. Special thanks go to Mrs. Vivian Alsip and Mrs. Gayanne Caprenter for their advice and support. Also, I want to thank Cathy Gallion for an excellent job of typing.

Finally, I want to thank my lovely wife, Li, for her love, patience, and understanding throughout this long undertaking and my great parents for providing me the chance to receive the best education.

Table of Contents

1.	INTRODUCTION	1
1.1	A Brief Survey of Multiprocessor Systems	1
1.2	Comparison of Three Multiprocessor Systems	4
1.2.1	PRIME System	5
1.2.2	C.mmp System	10
1.2.3	NonStop System.....	14
1.2.4	Overall Comparison	17
1.3	Major Design Questions	22
1.3.1	Software Related Questions	22
1.3.2	Hardware Related Questions	28
1.4	Thesis Outline	30
2.	SYSTEM PERFORMANCE MEASUREMENT	32
2.1	Queueing Analysis	32
2.1.1	Our Queueing Model	33
2.1.2	Avi-Itzhak and Heyman's Method	37
2.1.3	Konheim and Reiser's Method	43
2.1.4	Brown, Browne and Chandy's Method	50
2.2	Simulation	56
2.2.1	Memory Bandwidth Problem	57
2.2.2	The Simulator	64
2.2.3	Definitions of System Measurements	74
3.	EXPERIMENTAL RESULTS	76
3.1	Results for Software Related Questions	76
3.1.1	The Workload	77
3.1.2	Monoprogramming versus Multiprogramming	85
3.1.3	Memory Allocation Schemes	97
3.1.4	Job Scheduling Algorithm	121
3.1.5	Effects of Job Characteristics	128
3.2	Results for Hardware Related Questions	139
3.2.1	Hardware Quantity Effect	143
3.2.2	Hardware Speed Effect	159
3.2.3	Partial Connection	165
4.	CONCLUSION	191
4.1	Summary	191
4.2	Some Design Problems	203
4.2.1	Address Interleaving	203
4.2.2	I/O Connection	217
4.3	Further Problems	223
	References	225
	Appendix A	230
	Vita	235

Chapter I

INTRODUCTION

1.1 A Brief Survey of Multiprocessor Systems

The advent of large scale integrated circuit has created a tremendous impact on computer system design. In particular, LSI technology has made great strides in the areas of memory packaging and microprocessor design. The existence of inexpensive but powerful microprocessors and extremely high density semiconductor memory chips have led people to consider the design of large computer systems incorporating a large number of processors and memory modules.

In fact, the idea of using multiple processing units to handle various functions of the whole system is not new. People started thinking and building machines with multiple PEs at least twenty years ago. Back in 1958, Unger [1] designed a machine to perform pattern-recognition processing, which consisted of a central control computer and a processing element array. During the same year, three other systems were designed and manufactured, namely, National Bureau of Standards' PILOT system [2] and USAF's AN/FSQ-31 and 32 air defense systems. Although these old machines do not quite fit into the definition of multiprocessor system commonly accepted by people today (since some of them we prefer to call Multiple-Computer Systems), they do show the approach people used to speed up their systems, i.e., using several processing units to carry out several operations at the same time.

Before we go on, we would like to present a definition of a multiprocessor in order to clarify some ambiguities. According to the American National Standard Vocabulary for Information Processing [3], a multiprocessor

system is defined as "a computer employing two or more processing units under integrated control." A better definition was proposed by Enslow [4].

He defines a multiprocessor to be a system with:

- Two or more processing units
- Shared common memory
- Shared I/O channels, control units, and devices
- Single integrated operating system
- Hardware and software interaction at all levels

We will use this definition throughout this report. Obviously, a group of computers connected by some communication means, such as ARPANET which does not have all these five characteristics, does not qualify to be and will not be called a "multiprocessor" system.

So, the first "true" multiprocessor under this definition should be Burroughs' D-825 system [4,5,6,7] announced in 1960. A lot of multiprocessor machines have been designed and built since then, for example, Burroughs B-5000, IBM 704X/709X, CDC 6600, Univac 1108/1110, etc. A complete list can be found in [5], and a very good bibliography in [8].

Most of the multiprocessor systems only have a small number of processors, say 2 to 10. This is not surprising because they were built before LSI became popular and the hardware was still very expensive. Only a few machines were designed to have a large number of processing units. The most famous one is, of course, ILLIAC IV as well as its two predecessors, SOLOMON I and SOLOMON II, which were designed by Slotnick, et al [9,10] to work on problems involving differential equations, linear algebra, and weather data processing. Since all these problems contain a lot of matrix operations, sometimes thousands by thousands, they do need a machine with a large array

of PEs in order to get a reasonably fast response time.

In the past few years, the tremendous improvement in circuit performance and the drastic reduction in hardware price have made the multiprocessor design even more attractive. In particular, the advent of the LSI microprocessor has brought the system designer into a new world. People start building systems by using tens, hundreds, or even thousands of cheap but very powerful microprocessors. Recently, several projects have been proposed, e.g. [11], to construct systems with 1024 or more processing elements. Only the very low cost, say a few hundred dollars per PE, can make this kind of design possible. This was still a dream even five years ago.

Of course, a lot of questions arise in this kind of new design. For example, how do we interconnect so many processors, how do these processors share resources and communicate with each other, how do we control the operation of the whole system and fully utilize the hardware. Needless to say, all these questions need to be answered satisfactorily before we can come out with a good design. People are getting more and more concerned with these problems. It is our intention to make a thorough study of these problems in order to get a better understanding of how to design such a multiprocessor system.

Before we try to answer those questions, we would like to briefly discuss three well-known systems to give readers an idea of what kind of system we are dealing with. They are: PRIME system at the University of California at Berkeley [12], C.mmp system at Carnegie-Mellon University [13], and Tandem 16 NonStop system by Tandem Computers, Inc. [14].

All these systems are made up with a certain number of microprocessors and memory modules. However, due to a different set of design objectives,

e.g., degree of resource sharing, expandability, etc., each system has a completely different architecture and operating system design philosophy. For example, they use three fundamentally different interconnection schemes to connect their processors and memories, namely:

- Multiport memories
- Crossbar switch
- Time-shared common bus

We would like to list their differences, and try to compare their advantages and disadvantages. Hopefully, we can learn something from this study which can be used as a valuable design guide in the future.

1.2 Comparison of Three Multiprocessor Systems

1.2.1 PRIME System

The PRIME system is a medium-size, general-purpose time-sharing system whose design is aimed at improving the cost/performance ratio, reliability and privacy of current time-sharing systems. Figure 1 shows the system architecture of the PRIME system. It consists of five Meta 4 microprocessors (by Digital Scientific Corporation) and thirteen four-port memory modules. Every processor is connected to eight memory modules via a dedicated processor bus, so it is a multiport memory connection. (Since each processor only connects to about two-thirds of the total memory, we will call this a "partial" connection in later discussions.)

Meta 4 is a microprogrammable microprocessor that has a processor cycle time of 100 ns. It operates on 16-bit operands and has a 32-bit micro-store. The MOS memory is 16 bits per word, which has a 400 ns access time and a 600 ns cycle time. Each four-port memory module has 8K words made up from two 4K word submodules. There is a four-by-two switch inside each

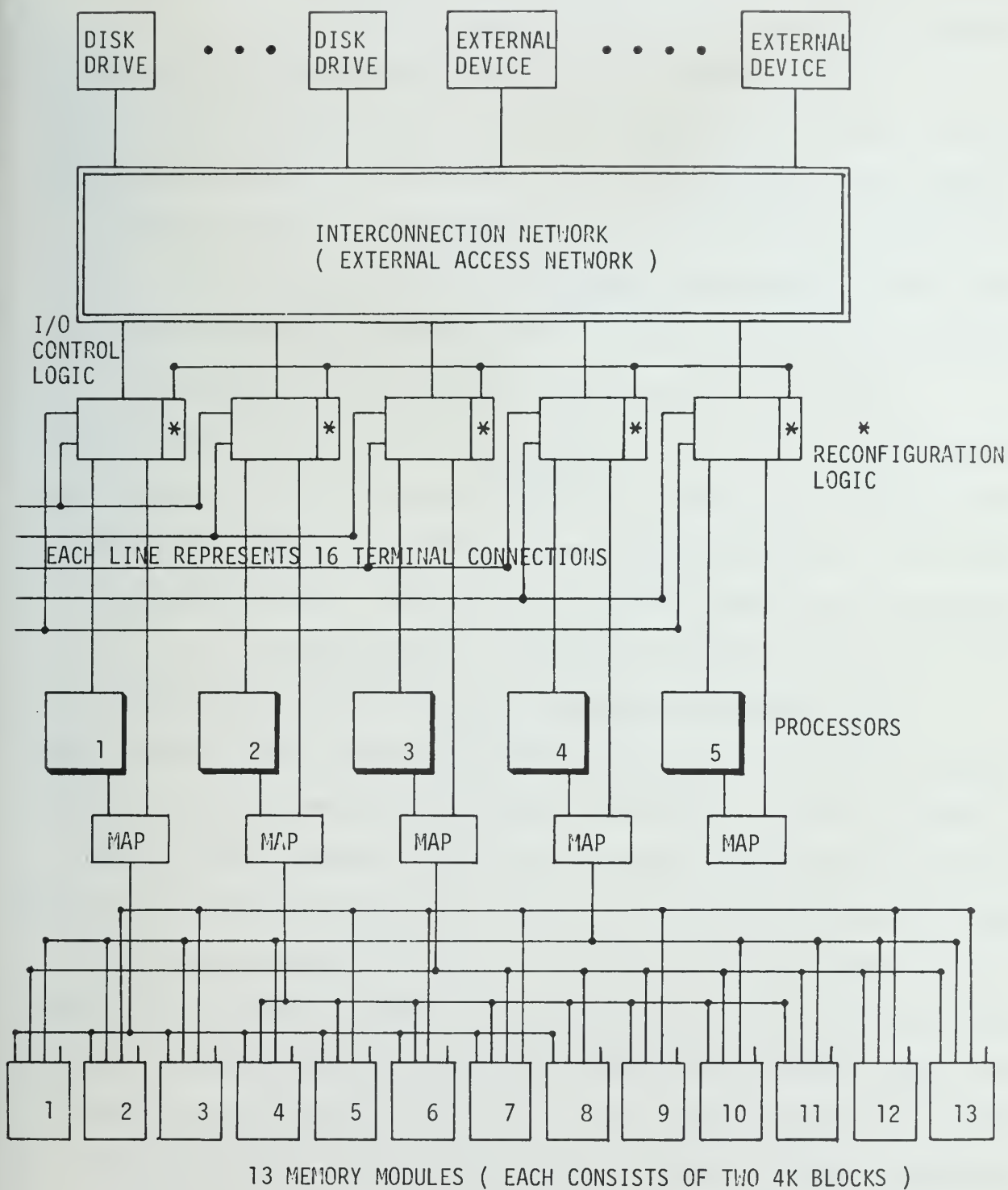


Figure 1. Structure of the PRIME System.

module which can connect a memory port to any submodule. This memory organization allows two-way interleaving inside a module.

All the peripheral devices (except user terminals) are connected to the processors and the memories via a big interconnection network called External Access Network (EAN), which is essentially a crossbar network [15]. The network is controlled by five I/O control boxes. These I/O control logic units not only control the information flow in and out of the peripheral devices but also control some inter-processor communication.

At any given time, the whole system is partitioned into five physically separated subsystems [16]. No two subsystems will share the same memory module or disk space. This is to achieve high privacy, which is very important in a multiprocessor system. One of these five subsystems is assigned to be the "control subsystem," and the rest "program subsystems." All the users will compete for these four program subsystems.

The operating system [17] is also partitioned into two parts, namely, the Central Control Monitor (CCM) and External Control Monitor (ECM), as shown in Figure 2. The control subsystem is assigned the Central Control Monitor and each program subsystem has a copy of External Control Monitor. CCM is the centralized part of the system wide operating system which controls all the system tasks like job scheduling, resource allocation, interrupt handling, and inter-processor communication. CCM also monitors the I/O control boxes to determine the connections made in the interconnection network. Whenever a program processor wants to talk to the other program processor or access a peripheral device, it must send a request to CCM and seek the permission from CCM. Then, CCM will make the connection by telling the

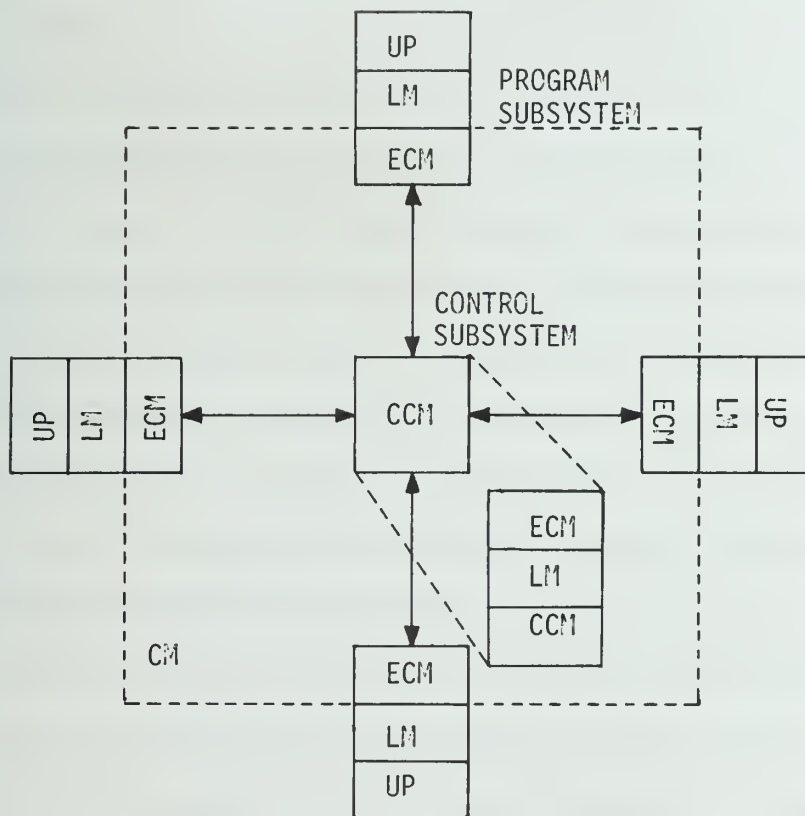


Figure 2. Structure of the PRIME Operating System.

interconnection network to do so.

ECM, on the other hand, is the local representative of the operating system at each subsystem, including the control subsystem. It will perform the local management functions related to processes running on that subsystem, e.g., teletype I/O for the teletypes physically connected to the subsystem's processor, swapping out the current process and swapping in the next process, etc. It also controls the communication between user processes and the CCM, and does the independent verification of CCM decisions. In fact, all five ECMs are working like a communication subnet.

Each subsystem also has a Local Monitor (LM). Every user can define his own LM to control all the intra-subsystem tasks, e.g., the management of resources allocated to him the generation of interrupts, etc. So, the software is modularized and partially distributed into all the subsystems. This is a very important factor for achieving high availability.

For reliability reasons, any subsystem can become the control subsystem. Whenever there is a failure in the current control subsystem (which may be detected by other subsystem's ECM), any other subsystem can take over the job immediately. If one program subsystem goes down, the whole system only suffers a performance degradation of 25%.

Hence, the PRIME system is a highly reliable, highly secure, and highly available system. Besides, due to its multiport memory connection, it is very easy to expand and reconfigure.

Of course, the physical boundary between two subsystems essentially eliminates the possibility of code sharing by two processes. Thus, some software duplication is needed which effectively reduces the available memory in each subsystem. However, the designers do not consider this as a

drawback. A paper by Ravi [18] points out that code sharing will actually generate more cons than pros, e.g., the system will need higher memory bandwidth due to the higher memory interference caused by competing processes.

There are two very interesting things we would like to point out. First, each program subsystem, and hence each processor, is dedicated to one user job until this job is swapped back onto the disk. A user job will not be brought into the main memory unless there is a free processor and the available memory space attached to it is large enough. So, at any given time, at most four user programs reside in the main memory being executed. In other words, the PRIME system does not allow more than one job to be executed by a processor subsystem at the same time. It would seem that they are not fully utilizing the processors. However, there is no overhead due to changing of jobs, e.g., swapping out the status information of the current job and reconfiguring a new subsystem. Furthermore, the operating system is much simpler which increases the software reliability.

The second thing is the partial connection scheme. The physical partition sometimes will eliminate the chance for a new job to enter the system, even though the total free memory space is large enough and there is some processor available. Of course, there is some amount of performance degradation due to this fact. We are interested in finding out how bad this is, compared to a more expensive full connection scheme like a crossbar switch.

Needless to say, the PRIME system does provide us a lot of interesting subjects to study. We will discuss them in more detail later.

1.2.2 C.mmp System

C.mmp is the multi-mini-processor system at the computer science department of Carnegie-Mellon University [13,19]. The overall architecture is shown in Figure 3. It consists of sixteen PDP 11 minicomputers connected through a crossbar switch to sixteen memory modules. Every processor (Pc, a modified PDP 11 processor) can access any memory module via the crossbar switch. So, the memory is completely shared by all processors. This is one basic difference between PRIME and C.mmp. We will call this kind of connection a full connection.

C.mmp is designed for solving large artificial intelligence problems. This kind of problem needs a number of processors to work on a large common data base simultaneously in order to obtain the answer in real time. So, complete memory sharing is crucial, and that is why it uses a 16×16 crossbar switch for the memory-processor connection. Of course, there will be some memory contention due to memory sharing. In the next chapter, we will give an analytical solution for this problem. However, the memory contention can be reduced by using local (or private) memory. In C.mmp, each processor has a 4K local memory which is not shared by other processors (Figure 3).

Each processor can be a slightly modified version of any model in the PDP 11 family. In the first stage of implementation, five PDP 11/20's were installed. Another four PDP 11/40's were scheduled to be added in the summer of 75. The PDP 11/40 operates on 16-bit operands and has a processor cycle time of 650 ns. Notice that, although the processor cycle time of PDP 11/40 is much larger than that of Meta 4, it does not mean the PDP 11/40 has a smaller instruction execution rate. The instruction execution rate

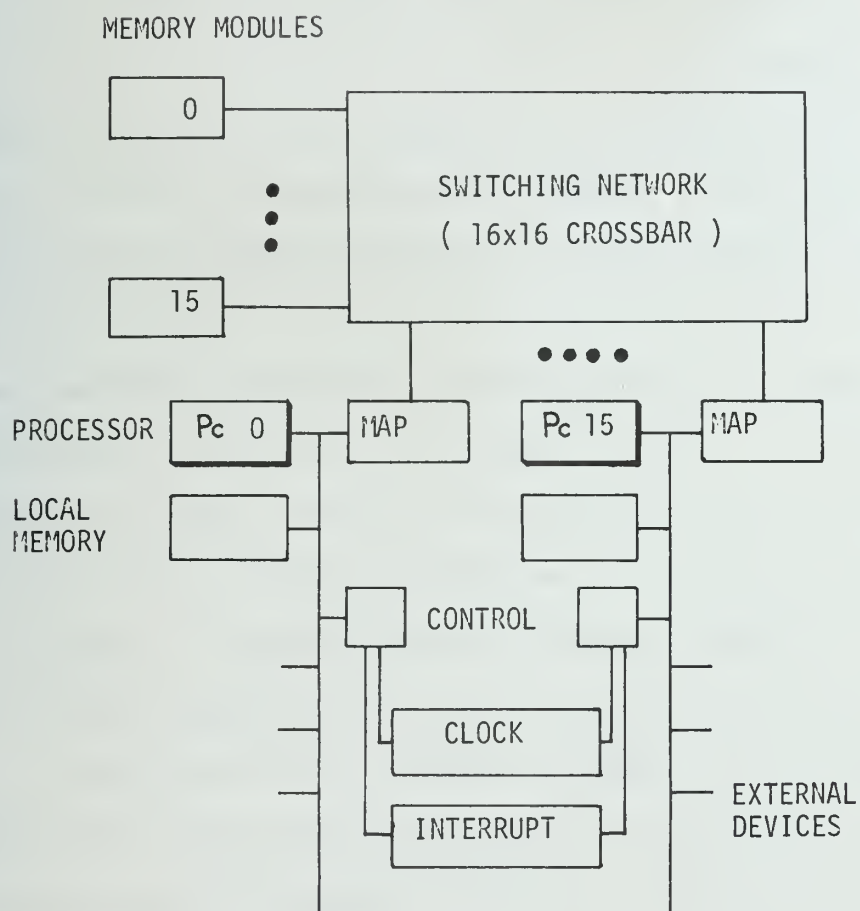


Figure 3. C.mmp Architecture.

is determined by the number of cycles each instruction will take. For PDP 11/40, most of the instructions only take one or two processor cycles. It averages roughly 0.44 million instructions per second. On the other hand, Meta 4 has the similar rate since each instruction needs several micro-instructions to execute.

Both PDP 11/20 and 40 use core memory which has a 500 ns access time and 1.2 μ s cycle time. Although the memory speed is not very fast, we can interleave a program into all 16 modules to get a high memory bandwidth.

One other area where C.mmp differs from PRIME is that peripheral devices are not shared. Each peripheral device is connected to the unibus of a processor and can only be used by this processor. Hence, the processors must use the primary memory for interchanging information. Both this and memory sharing are possible sources of violating privacy. Software protection is a very important issue in the operating system design.

Although the main purpose of C.mmp is to use the system as a whole to work on a large program, it can also be partitioned, either dynamically or statically (manually), into several independent subsystems and operated in a fashion like PRIME. Due to the partitionability of the cross-bar switch, the hardware can be partitioned into two, three, or even 16 totally separated subsystems. This provides a great ease for maintenance, since if any processor or memory module is down, it can be isolated from the rest of the system and turned over to the hardware engineer for replacement. Thus, it would not require taking the entire machine down for maintenance.

Unlike the PRIME system, C.mmp does not designate a single

processor as the control processor. This is because C.mmp is designed to have up to 16 processors, and when the number of processors increases, the master (or control) processor quickly becomes a bottle-neck. This is the reason why PRIME can only have a small number of processors (5), since Meta 4 is a relatively slow minicomputer. However, this means that each processor in C.mmp should have its own copy of the operating system if it is working alone. In order not to occupy too much memory, the size of the operating system should be somehow minimized but still meet all the users' requirements. Certainly, this needs a special kind of operating system design.

HYDRA, the operating system for C.mmp, is designed for this purpose [20]. The central core of HYDRA is a "kernel" set of operating system facilities which provide both basic protection and management of the hardware resource. However, the kernel does not provide software for things like the file system, job control language, or scheduling policy. These are supplied by the user.

This approach has several advantages. First, the user has the freedom to define his own operating system, for example, a job control language. This not only allows the user to minimize the size of his operating system, but also allows him to specify some facility not provided by the existing programs or to replace an existing facility by one more closely attuned to his own needs. Second, an error in one user operating system can only affect his own program. It will not crash the entire system. This greatly increases the reliability of the software. Since the kernel is rather small and well-defined, an error is very rare.

In the C.mmp system, a program usually can be run on any available processor. This is why it needs a crossbar switch in order to provide a full

connection. Of course, the processor utilization will be higher than that on PRIME. However, we can see there must be a big overhead associated with job swapping, especially when each user has defined his own operating system. We will show later that this scheme might not be a good idea in some cases.

The use of a crossbar switch, of course, has some disadvantages: first, it is very expensive; second, it is not easy to expand. C.mmp can have a maximum of sixteen processors and sixteen memory modules. Although if we use PDP 11/40s, the system can yield up to 7 million instructions per second (mips), which is comparable to an IBM 370/158, it will be very difficult and expensive to expand the system beyond sixteen processors. This scheme certainly will not work in future systems where we might have thousands of processors.

But, in general, C.mmp is a very reliable, both in software and hardware, highly available, and easy to maintain system. In particular, the ideas of HYDRA will be very helpful in operating system design for future multiprocessor systems.

1.2.3 NonStop System

Figure 4 shows the architecture of a recently announced multiprocessor; Tandem Computers' NonStop System [14]. This system, configured with up to 16 minicomputers, is designed to handle heavy banking transaction processing and provides very high availability. The basic difference is that the processors are connected together by a pair of time-shared common buses. Processors will communicate with each other via these buses.

The use of common bus offers the advantages of very low cost and

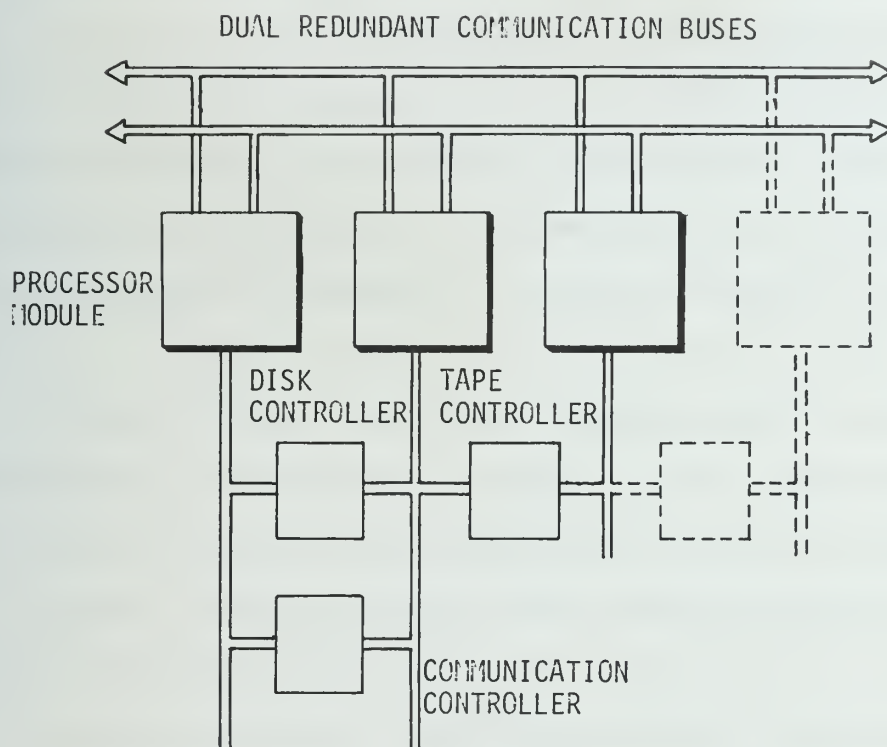


Figure 4. Tandem 16 NonStop System.

the ease to modify the hardware configuration. For example, we can add or remove a functional unit fairly easily. However, the overall system performance is limited by the bus transfer rate, and the failure of the bus will cause a catastrophic disaster. Hence, NonStop uses dual redundant buses to increase the transfer rate and the availability.

Each processor module is actually a complete minicomputer, having its own control unit, arithmetic unit, private memory, and its own copy of the operating system. So, every processor has the ability to keep working even if all other processors are down. Also, whenever a processor goes down, other processors can take over without much difficulty.

There is no memory sharing, instead the processors share the peripheral devices (e.g., disk and tape) via the controllers. This is because the system is designed for handling banking transactions and all the processors are supposed to work on a big data base on disks or tapes. Therefore, unless each minicomputer can provide a large amount of primary memory, the use of this kind of architecture is perhaps only appropriate for data base management.

The number of processor modules that can be attached to the common data bus seems to be unlimited. However, as the number of processors increases, the bus contention increases drastically. This will seriously degrade the system throughput. Besides, the longer the bus is, the larger the time skew is, and the slower the clock rate will be. So, the expandability is limited by a number of constraints. When the workload grows beyond a certain limit, this architecture will no longer be able to expand and perform satisfactorily.

The I/O is controlled by the communication controller. The

controller can assign a task (transaction) to any available processor. This can achieve a high availability and utilization of processors.

Perhaps the most appealing design is their software. Guardian, the operating system of NonStop, is a virtual memory system which contains automatic re-entrant, recursive and shareable codes. Whenever a component fails, Guardian automatically reassigns both processor and I/O resources to ensure that in-process tasks including file updates are completed correctly. This guarantees the process can be restarted in a very short time. For a system that provides high availability, this type of action is extremely important.

When one of the disks fails in the middle of a file update, Enscribe, Tandem's NonStop data base manager, ensures that the damaged record or file is restored. Enscribe uses a duplicate file technique to continue the operation by using the back-up file. Hence, the faulty disk will not cause any interruption of service.

Overall, NonStop uses redundant hardware and duplicated software in order to give the user continued service without any interruption or termination. This is very important to a system where the user cannot afford any system downtime, for example, a telephone switching network, or an online banking situation. However, this might not be a good candidate for a scientific research environment.

1.2.4 Overall Comparison

We went through three multiprocessor systems very briefly in the last three sections. As we can see, each system has a different architecture, and its own advantages and disadvantages. It is not fair to say which system is the best or which one is better than the other one since they have

Features	PRIME		C.mmp	NonStop
Size	Medium	Large		Medium
Interconnection Scheme	Multipoint Memories	Crossbar Switch		Time-shared Common Data Bus
Main Memory Sharing	No (after partition)	Yes		No
Peripheral Device Sharing	Yes	No		Yes
Contention	Only I/O	Main Memory		Common Data Bus
Physical Boundary Between Subsystems	Yes	No		Yes
Operation Mode of the Operating System*	Master-Slave	Symmetric Processors		Separate Executive
Operating Environment	Monoprogramming	Multiprogramming		---
Ability to Define Own Operating System	Yes	Yes		No
Hardware Reliability	High	High		Medium (System will fail apart if both buses fail)
Software Reliability	Low (Failure of Control Processor might cause disaster)	Low (Memory Sharing might break the privacy)		High
Availability	High	High		High
Maintainability	Easy	Low		Medium
Software Complexity	Simple	Complex		Complex
Number of Processors Designed in System	8	16		16
Hardware Bottleneck	Microprocessor	Crossbar Switch		Common Data Bus
Degree of Software Duplication	None	Low		High
Means for Inter-processor Communication	External Accessor Network**	Main Memory		Common Data Bus
Design Goal	To support a large number of time-sharing terminals	Solve large A.I. problems in real-time, also can support a number of users simultaneously		Heavy Banking Transaction Processing
Cost ^Δ	Low	Medium		Medium

Table 1. Summary and Comparisons

* See the definitions in [5].
 ** A partial crossbar, see [15].
^Δ Hardware costs, relative to large comparable systems, e.g., 370/158.

different design goals. However, they do provide us with a lot of interesting problems to look at.

In the next section, we will list a number of problems we want to study. Before we do that, we will give a comparison of these three systems in Table 1. But first, let us give the definitions for the terms "monoprogramming" and "multiprogramming" we use in Table 1. These two terms will be used very often in our later discussion.

In a single processor system, or sometimes called a monoprocessing system, the distinction of these two terms can be easily made. If at most one job can be in execution in the system, i.e., the system only executes one job at a time, it is in general called a monoprogrammed system. In fact, it should be called a monoprocessing, monoprogrammed system. Many minicomputer systems belong to this category. On the other hand, if more than one job can be in execution at the same time, it is called a monoprocessing, multiprogrammed system. Of course, the advantage of a multiprogrammed system is that the processor can execute a job while the other jobs are doing I/O. This can increase the processor utilization and in general shorten the turn-around time for a job. Almost all the large systems are operating in this mode, e.g., our IBM 360/75 system.

In a multiple processor system, however, the situation is more complicated. Of course, if the system still only allows one job in execution, i.e., all the processors are working on one job, it is called a multiprocessing, monoprogrammed system. A SIMD machine like Illiac-4 belongs to this category, where all the processors are under the control of one single instruction stream. The C.mmp system sometimes also operates in this mode when all the PDP 11's are running one giant artificial intelligence program. In that case, it should be considered as a multiprocessing, monoprogrammed

system. However, when more than one job is allowed in a multiprocessor system, which is the case in the system we will deal with, the classification is more complicated and sometimes rather confusing. For example, it is difficult to classify a system where several processors are executing one job while many other jobs are running on one other processor. If we follow the way the other three kinds of systems are classified, this kind of system should be called a multiprocessing, multiprogrammed system. But, multiprogramming is usually used for a system where each processor is able to handle more than one job. It is rather unfair to call all the multiprocessor systems that allow more than one job to be executed at the same time multiprocessing, multiprogrammed systems.

Fortunately, we will only talk about this kind of system, i.e., a multiprocessor system that allows several jobs to be executed at the same time. So, we do have some freedom to use these two terms and can avoid much confusion.

In the rest of this thesis, we will use monoprogramming to mean a system that allows each processor to have at most one job at any given time. That is, each processor will be dedicated to a certain job throughout the lifetime of this job and cannot execute any other jobs. While a job is doing I/O, its corresponding processor will be idling. From a processor's standpoint, it seems that there is only one job in the memory. This is why we use the term monoprogramming even though there might be more than one job executing in the whole system at the same time. The PRIME system falls into this category and hence will be called a monoprogrammed system. A consequence of using monoprogramming is the number of jobs in execution will never be larger than the number of processors in the system.

On the other hand, multiprogramming will be used to indicate a system that allows more jobs than processors to be in the memory at the same time. Therefore, each processor might be responsible for more than one job at a certain time instant. In a multiprogrammed system, there need not be any linking between any job and any processor. All the jobs can be thought to be in a "pool." Whenever a processor is free, it will grab a job in the pool to execute. The C.mmp system sometimes operates in this mode, so we classify C.mmp as a multiprogramming system in Table 1.

Although all these systems have quite different software and hardware structures, as we can see from Table 1, they all have the following advantages: high availability, high hardware reliability, high flexibility, relatively low cost, and ease of maintenance. These are the main design objectives of the multiprocessors. Most of all, they all provide very high throughput due to concurrent operation of the processors.

However, the multiprocessors also have some disadvantages. For example, the ability to expand a system is usually limited by the hardware capabilities like the processor speed, bus transfer rate, etc. In addition, the system software is rather complicated, which makes it difficult to design, expensive to produce, difficult to check out, and difficult to maintain [4,5]. The design of a simple, self-checking operating system should be an important subject in multiprocessor design. We will not go into this subject here. However, we do want to study the performance tradeoffs of some simple strategies, for example, monoprogramming versus multiprogramming. These results will be shown in Chapter 3 later.

1.3 Major Design Questions

In this section, we will describe all the questions we are interested in. Hopefully, this can give the reader a complete idea of what we are after. Although these questions only cover a subset of problems a system designer must be concerned about, we do believe they are basic and important issues in designing a multiprocessor system. We will roughly subdivide these questions into two groups, namely, software and hardware related questions, and describe them separately.

1.3.1 Software Related Questions

Almost all the software questions we list here affect the operating system design. As we just mentioned, a simple and reliable operating system is a very important aspect in multiprocessor design. So, our attention will concentrate on the simplicity and the reliability of an operating system.

From what we have learned in the previous sections, naturally, our first question will be:

- Should we use multiprogramming or monoprogramming?

Intuitively, multiprogramming can result in higher utilization of both processors and memories, since it allows more jobs to be packed in the memory and the overlapping of processor and I/O operations. Of course, the busier the processors are, the more work they do, and the higher throughput they produce.

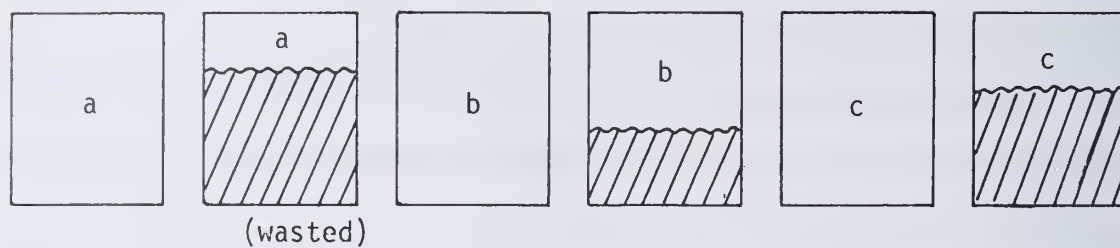
However, this does not necessarily mean the turnaround time of a job will be better than monoprogramming. There will be higher overhead due to multiprogramming. For example, the system needs to save the current status of the pending job in the memory and bring in the status information

of the new job. Moreover, resource contention will increase since more jobs are being executed at the same time. All these will certainly increase the service time of a job. Although multiprogramming may yield shorter queueing time, the total turnaround time will not necessarily be shorter.

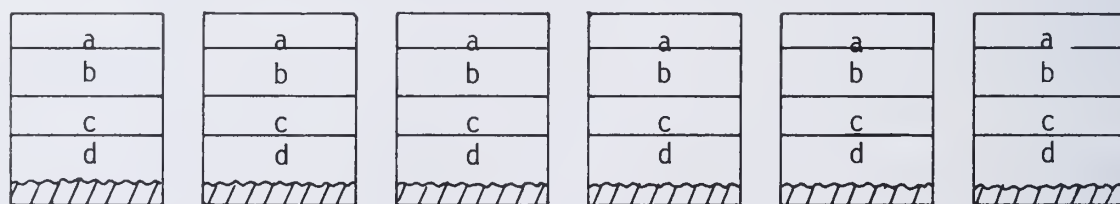
As we pointed out earlier, the biggest disadvantage of multiprogramming is that the software is more complex, expensive, and fault-prone. Unless it can outperform monoprogramming by a large margin, there is no reason why we should use it. In Chapter 3, we will measure the performance of both systems, which can tell us which system is better to use in a multiprocessor system.

Our second question is about the memory allocation. When a job comes in, we can, as in the PRIME system, allocate as many modules as it needs. This job will occupy these modules exclusively, and no sharing is allowed (Figure 5-a). We call this scheme a "partitioned" scheme. Of course, no interference between two jobs will occur, since jobs are physically separated. Therefore, a partitioned system is very reliable and provides high privacy.

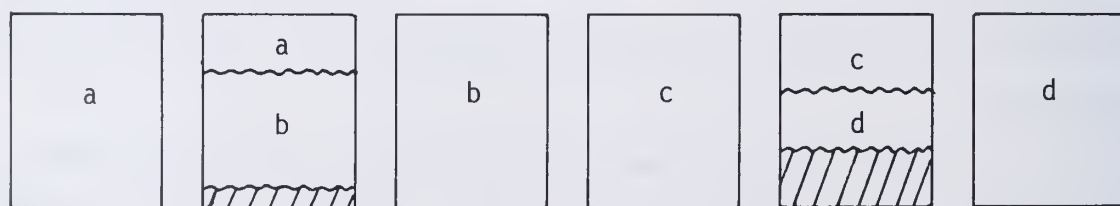
One disadvantage is that a lot of memory space will be wasted. On the average, each job will waste about half of a memory module if the job size is uniformly distributed. This can be shown in a few lines. Assuming each memory module is of size x , and the job size is y , then the expected wasted memory is:



(a) Partitioned System



(b) Distributed System



(c) Mixed System

Figure 5. Memory Allocation Schemes.

$$\begin{aligned}
 x - E(y - \lfloor \frac{y}{x} \rfloor * x) &= x - \sum_{i=0}^{x-1} i * \Pr(y = \lfloor \frac{y}{x} \rfloor * x + i) \\
 &= x - \frac{1}{x} * \frac{x(x-1)}{2} \\
 &= \frac{x+1}{2}
 \end{aligned}$$

If there are j jobs in the memory, then at least $\frac{j(x+1)}{2}$ memory space is wasted on the average. Even more memory will be wasted if the number of modules left over is not large enough for the next job. So, this scheme might not be a good idea unless we can use small memory modules.

The other extreme, as in C.mmp, is to allow the jobs to share all the available space. One way to share memory is to distribute a job across all the memory modules, as shown in Figure 5-b. This scheme we call a "distributed" scheme.

In a distributed system, we can pack as many jobs in the memory as possible, and each job can enjoy a maximum degree of interleaving. However, the memory contention problem arises due to the memory sharing. Moreover, the failure of any single module can cause the collapse of the whole system.

Of course, we can combine these two schemes and form a third scheme, which we call a "mixed" scheme (as shown in Figure 5-c). Basically, this is a partitioned system, except the "overflow" parts are put together in a single module. Thus, we can pack more jobs in the memory, reduce the wasted space, and still enjoy most of the advantages offered by a partitioned system. We summarize the advantages and the disadvantages of these three schemes in Table 2.

Scheme	Advantages	Disadvantages
Partitioned	a. More reliable, the failure of one memory module will only affect one job. b. Easy to expand and maintain, the addition or deletion of a module will not affect the operation of the whole system. c. No inter-subsystem interference.	a. A significant amount of memory is wasted, cannot fully utilize the memory. b. Cannot allow as many jobs in the memory as the other schemes. c. Can only interleave a program in its own modules.
Distributed	a. Can fully utilize the whole memory. b. Allow each program to have a maximum degree of interleaving.	a. Not very reliable, the failure of one module will crash the whole system. b. The addition or deletion of a module will affect all programs. c. Have inter-subsystem interference.
Mixed	a. Can fully utilize the whole memory. b. The failure of one module will only affect a few programs. c. Inter-subsystem interference is small and can only occur in those shared module.	a. Interleaving will be a problem if a program must share a module with other program.

Table 2. Comparison of Three Memory Allocation Schemes.

Now, our second question is:

- What kinds of performance will be given by these schemes, and which scheme is the best to use?

In Chapter 3, we will measure all these schemes, and discuss some problems related to them, for example, how do we interleave the addresses if we want to use the partitioned system or the mixed system.

The next question we are interested in is:

- What kind of scheduling algorithm should we use?

Every operating system designer must face this question. In order to answer it, we have to know what type of system we are dealing with, what kind of measurement we are interested in, and what penalty we will suffer if we let some job wait in the queue. In general, people measure the goodness of a scheduling algorithm by using the average turnaround time it produces. So, most people use round-robin in a time-sharing system, shortest-job-first in a batch system, and give higher priority to time-sharing jobs in a mixed system.

In our study, we will assume that we are dealing with a batch system. However, this does not imply the shortest-job-first (SJF) algorithm will always win if we consider the average turnaround time, since we will deal with some special architectures like a partially connected system. Some other algorithm, e.g., the best-memory-fit-first, might perform better than SJF under that circumstance. We are even interested in seeing how bad the first-come-first-serve algorithm will perform, since FCFS does not involve any scheduling and that means a simple operating system.

The fourth question we want to answer is:

- How do the job characteristics affect the system performance?

The job characteristics include the mean, the variance, and the distribution of the job size, the processing time, the inter-arrival time, and the number of I/O requests. These parameters certainly have great influence on the system performance. For example, if the mean job inter-arrival time is too small, i.e., the jobs come in too fast, the system will become over-saturated and the queueing time might go to infinity.

In order to avoid all these undesirable phenomena, we must understand how the system responds when a certain parameter changes, and how sensitive it is. Then, we can always keep our system in a safe region. Whenever the system load changes, we will know what we should do in order to maintain satisfactory performance.

1.3.2 Hardware Related Questions

The cost-effectiveness is perhaps the subject people are most concerned about in system design. Every designer wants to know how much performance improvement he can get for a certain piece of hardware he adds. Of course, everyone will try to invest his money where he can buy the best improvement. So, this trade-off problem usually is the first thing people will solve.

Like most people, our first hardware related question is:

- How is the system performance affected by the architectural parameters?

We want to know how the job turnaround time, processor and memory utilizations vary when we change certain hardware parameters, like the number of processors, the number of memory modules, the number of I/O devices, or the size of a memory module, etc. We also want to know how the system performance will be affected by the hardware characteristics, like memory

cycle time, the processor cycle time, and the I/O speed.

As we mentioned in Table 1, the architectural difference between those three multiprocessor systems is the interconnection schemes they use, namely, the crossbar switch, the multiport memories, and the time-shared common buses. In fact, we can classify them into two groups according to the degree of connection each scheme provides. The crossbar switch used in C.mmp will be called a "full connection," since every processor can access any memory module via this switch. The multiport memories used in PRIME or the common buses used in NonStop, on the other hand, will be called a "partial connection," since every processor can only access part of the memory.

Naturally, we would like to know:

- How much degradation will we suffer if we use a partial connection instead of a full connection?

Of course, the degradation goes up as we decrease the number of connection points. But at the same time, we reduce the cost of the whole system. Obviously, this is a trade-off problem. We will do some comparisons in Chapter 3.

There is another very interesting problem associated with the partial connection, and we call it the "connectivity" problem. Since in a partial connection, say multiport memories, each memory module will be connected to only some of the processors, so the question is how many memory modules will be connected to a particular processor. For example in Figure 1, each memory module has four ports (but only three have been used, except one module uses 4), and each processor is connected to eight memory modules in a fairly regular manner. However, this uniform connection might not be

the best way. Especially when the number of ports is small relative to the number of processors, then some uneven connection might be necessary in order to meet some requirements, for example, one processor must be connected to half of the memory modules in order to take care of big jobs. Again, we will devote a section to discuss this interesting problem.

1.4 Thesis Outline

In order to answer the questions we raised in the last section, we need to do some performance measurements. Two methods are commonly used by people for measuring the system performance, namely, queueing analysis and simulation. Usually, queueing analysis can reveal more insight about the system behavior, since we can see from the analytic solution how a certain variable affects the system performance. In the first part of Chapter 2, we will discuss some analytic work people have done in measuring computer systems.

However, queueing techniques are only good for simple models with some simple assumptions. As the complexity of a system model increases, the queueing analysis will soon become intractable. Therefore, people switch to simulation. The nice thing about a simulation model is you can put in as many parameters as you want, and as many constraints as you like. So, the simulation technique can be applied to very complicated model. Unfortunately, it can be very costly.

Since our model is rather complicated, we will use a combination of the analytic approach and simulation to measure the relative performance of various systems. In the second part of Chapter 2, we will describe the simulation model we use. We will also talk about some memory bandwidth

problems, since we will use memory bandwidth to determine how we advance the virtual clock of our simulator.

In Chapter 3, we will present all the results and try to answer the questions we raised in the last section. Finally, in Chapter 4, we will discuss some logic design problems and give a summary of all our results.

Chapter 2 SYSTEM PERFORMANCE MEASUREMENT

2.1 Queueing Analysis

In this chapter, we are going to talk about how we measure the performance of a multiprocessor system. As we just mentioned at the end of the last chapter, two common techniques can be used for this purpose: queueing analysis and simulation. We will start by looking at some queueing models that have been proposed for analysis of multiprocessor systems.

Using queueing techniques to study the system performance is a very old subject. People have been active in this area for quite a number of years and a lot of papers have been written on this subject. However, most of the effort has been spent in the following areas:

- (1) Performance analysis of auxiliary and buffer storage like disk [21,22], drum [23], or magnetic-bubble [24].
- (2) Waiting time analysis of job scheduling disciplines [25,26].
- (3) Performance analysis of single processor multiprogramming systems [27] and single processor time-sharing systems [28].
- (4) Performance study of communication networks like ARPANET [26] or ALOHA [29].

Relatively few papers have been written about multiprocessor systems. Perhaps the biggest difficulty is to formulate the resource contentions into the model. Besides, if we want to consider the finiteness of memory size and I/O operations, then the whole system will become a queueing network with blocking. This is a well-known tough problem to be solved exactly. In some papers, e.g., [30], people just ignore all these problems and treat the multiprocessor as a M/M/p queueing system, which yields a bad

approximation. Of course, we are interested in a more accurate solution.

Recently, three papers have been written which provide some analytic methods of studying multiprocessor systems [31,32,33]. In two of these methods, we can accurately include the effects of finite memory size and workload memory requirements in the queueing model. We feel that these papers deserve to be discussed in some details in order to let readers have more understanding about the problem and realize the strength of these analytic methods. We will show how we apply these methods to our queueing problem, and discuss the advantage and disadvantage of each method. However, due to the complexity of the systems we are going to study, we will have some difficulty including all the effects of system architecture and resource contention into a queueing model. Unless we can nicely formulate everything, we will not be able to get accurate results from any of these methods. After we discuss these three papers, the reader should realize why we will rely on simulation rather than queueing analysis.

Before we talk about these analytic models, let us first describe our queueing model for a multiprocessor system. This will aid in understanding the later discussion.

2.1.1 Our Queueing Model

Figure 6 is the basic queueing model we are interested in. We assume that the system has p processors, r I/O devices, and a total of M kbytes of primary memory divided into m modules. When a job arrives, if either there are already D jobs in the system or the available memory is not big enough, it will be queued in the outside waiting queue. Otherwise, this job will enter the service box and queue in the processor queue for the first service. If the job gets a processor, it will be served for some amount

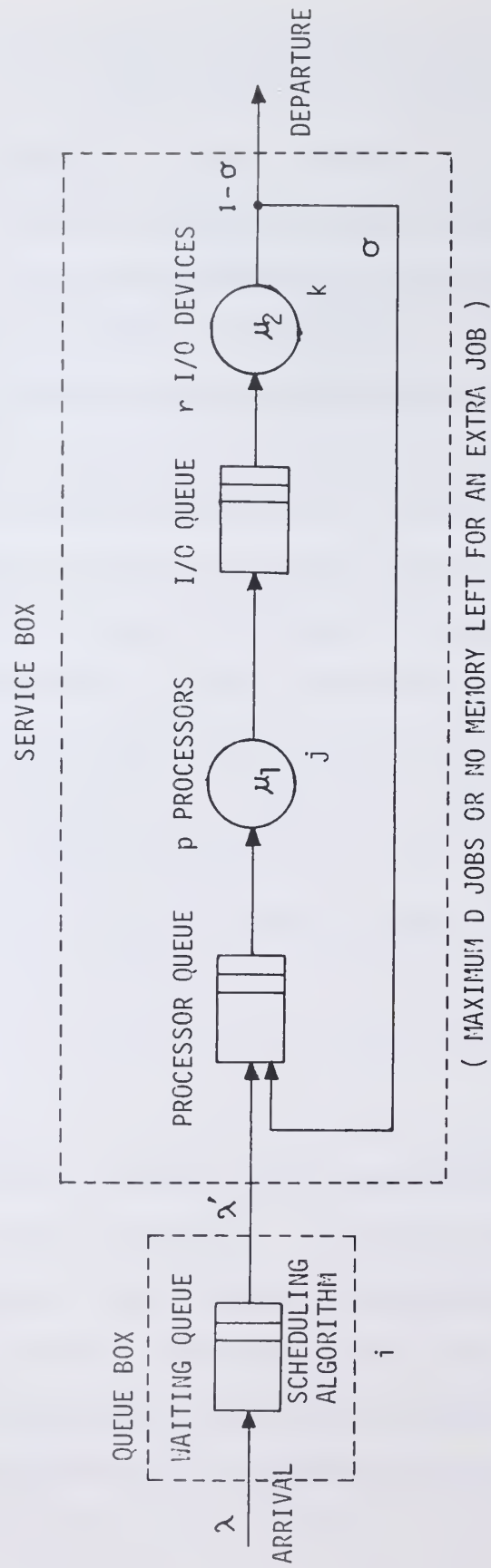


Figure 6. Our Queueing Model.

of time nonpreemptively until it requests an I/O operation. Then it proceeds to the I/O queue and waits for an I/O operation. After the I/O, this job will depart the whole system with probability $1-\sigma$, or return to the processor queue with probability σ and the cycle starts again.

The outside waiting queue corresponds to the HASP queue in our IBM 360/75 which holds the jobs that are blocked from service. The average number of jobs queued here is an indication of how well the system performs. A good scheduling algorithm could be used here to reduce the average queue length.

The number D indicates the maximum number of jobs allowed in the service box. It is equal to p under monoprogramming and some constant d under multiprogramming (d is called degree of multiprogramming).

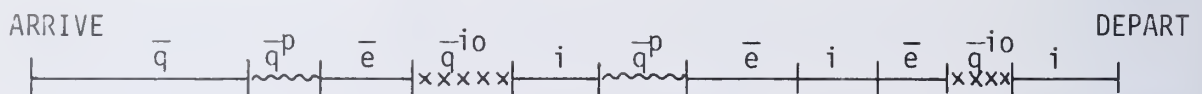
Each job in the service box will cycle through the tandem queue for a certain number of times. This number will have a geometric distribution:

$$\Pr \{ \text{a job needs } \ell \text{ cycles} \} = (1-\sigma)\sigma^{\ell-1}, \ell=1,2,\dots$$

with mean $\bar{a} = 1/(1-\sigma)$. $\bar{a}$ is actually the average number of I/O requests for a job. The parameter σ can be arbitrarily defined or obtained from analyzing some real data.

Figure 7 shows the timing diagram of a job from its arrival until its departure. Of course, in any stage, if the resource is available when the job arrives, it will get served immediately without waiting.

In order to make analysis easier, people always assume the job arrival is a poisson process. In other words, the job arrival rate is constant or the inter-arrival time is exponentially distributed. Literature on queueing analysis has shown evidence that this is a pretty acceptable assumption.



- $\bar{q}$: QUEUEING TIME IN THE QUEUE BOX
- $\bar{q}^p$: QUEUEING TIME IN THE PROCESSOR QUEUE
- $\bar{e}$: EXECUTION TIME
- $\bar{q}^{io}$: QUEUEING TIME IN THE I/O QUEUE
- i : I/O TIME

Figure 7. History of a Job.

However, the most controversial part is the service rates of processor and I/O, i.e., μ_1 and μ_2 in Figure 6. In order to use the nice results of queueing theory, we will have to assume they are constant, that is, to assume both the processing time and the I/O time are exponentially distributed. This is a very strong assumption to make.

For the I/O service rate, if we neglect the interference between I/O requests, this assumption may be alright. But, this cannot be true for the processor. The service rate of a processor should be a function of system architecture, memory allocation strategy, the number of jobs currently being executed, the memory interference they create, and the original inter-I/O time distribution. In general, this is not easy to formulate.

In addition, due to the finiteness of the memory size and the maximum number of jobs allowed in the system, this model becomes a queueing system with blocking. It is a tough problem and no exact solution is known yet [34]. The best thing people can do is to use an approximate model. One example is Avi-Itzhak and Heyman's model which we are going to discuss in the next section.

2.1.2 Avi-Itzhak and Heyman's Method

The approach adopted by Avi-Itzhak and Heyman consists of two stages [31]. The first stage is to view the system as a closed queueing network with a fixed number of jobs, and obtain the average cycle time for each job. Then we approximate the open system by an M/G/D queue, use the result of stage one to solve the state balance equations, and get the expected time a job will spend in the system.

By a closed queueing network, we mean a network with no job coming in or going out. Figure 8 shows the closed queueing network used in Avi-Itzhak and Heyman's analysis. It consists of k service stations with a fixed number, say n , of jobs cycling through the stations. Station 1 represents a group of processors, and stations 2 to k represent various groups of peripheral devices such as disks, tapes, etc. Station i contains r_i servers operating in parallel with a common queue, and each server has the same expected service time $E(S_i)$. When service at a processor is completed, the job moves to station i with probability π_i , where $\pi_1=0$ and $\sum_{i=2}^k \pi_i = 1$. Upon completion of service at station i , the job moves back to station 1 and the same process is repeated.

The exact solution of this queueing model has been obtained by Jackson [35], and Gordon and Newell [36]. We will repeat their result here and show how to relate it to our model.

Let p_i be the steady-state expected number of busy servers at station i . The average number of jobs flowing into a given station must equal the average flow out of the station. Therefore,

$$[p_1/E(S_1)]\pi_i = p_i/E(S_i), \quad \text{for } i = 2, \dots, k.$$

Define $a_i \triangleq [E(S_i)/E(S_1)]\pi_i$,

we obtain $p_i = a_i p_1$,

with $a_1=1$ by definition.

Assume $p(x_1, x_2, \dots, x_k)$ is the steady-state joint probability of there being x_i , $i = 1, 2, \dots, k$, jobs at station i , then we have

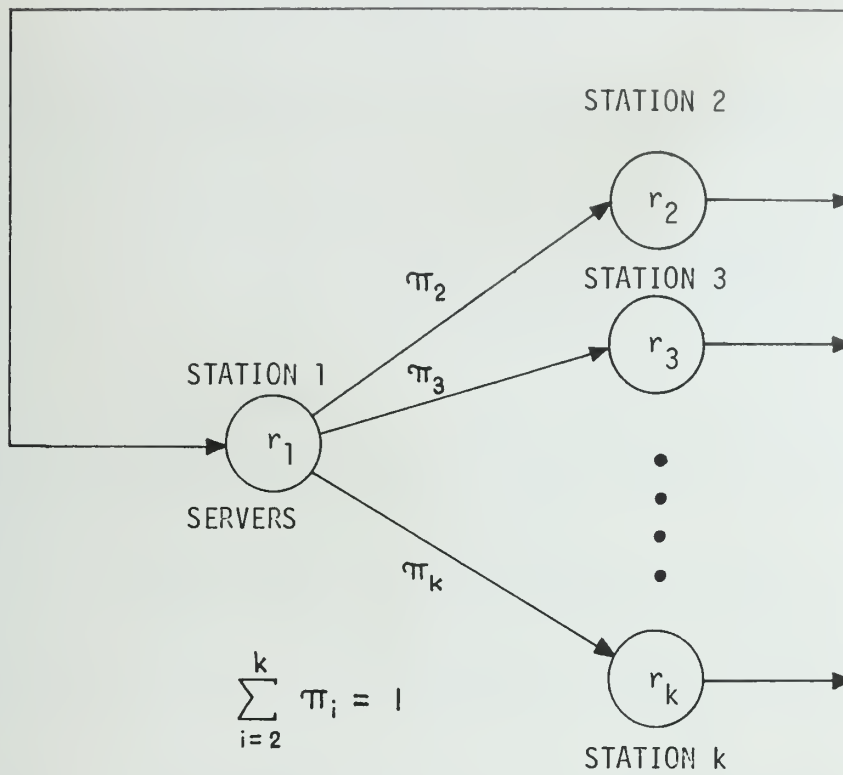


Figure 8. The Closed Queueing Network used in Stage One.



Figure 9. Approximated M/G/D Queueing Model.

$$p(x_1, x_2, \dots, x_k) = c \prod_{i=1}^k [p_i^{x_i} / \beta_i(x_i)]. \quad (1)$$

In this equation all $x_i \geq 0$, $\sum_{i=1}^k x_i = n$, c is the normalization constant, and

$$\beta_i(x_i) = \begin{cases} x_i! & \text{if } x_i \leq r_i, \\ r_i! r_i^{x_i - r_i} & \text{if } x_i > r_i. \end{cases}$$

The summation of $p(x_1, x_2, \dots, x_k)$ over the set $D_n = \{x: x \geq 0, \sum_{i=1}^k x_i = n\}$ must yield the value 1. Therefore from equation (1), we have

$$1 = c \sum_{D_n} \prod_{i=1}^k [p_i^{x_i} / \beta_i(x_i)] = c p_1^n \sum_{D_n} \prod_{i=1}^k [a_i^{x_i} / \beta_i(x_i)] \quad (2)$$

However, the expected number of busy servers at station 1 is p_1 , that is

$$p_1 = \sum_{x_1=1}^{r_1-1} x_1 \sum_{D_n - x_1} p(x_1, x_2, \dots, x_k) + r_1 \sum_{x_1=r_1}^n \sum_{D_n - x_1} p(x_1, x_2, \dots, x_k), \quad (3)$$

where $D_n - x_1 = \{x: x \geq 0, \sum_{i=2}^k x_i = n - x_1\}$. Substitution of (1) and (2) into

(3) yield

$$p_1 = \left\{ \sum_{x_1=1}^{r_1-1} x_1 \sum_{D_n - x_1} A + r_1 \sum_{x_1=r_1}^n \sum_{D_n - x_1} A \right\} / \sum_{D_n} A$$

with $A = \prod_{i=1}^k [a_i^{x_i} / \beta_i(x_i)]$.

The only thing we need now is an algorithm to generate all the elements in D_n and $D_n - x_1$, then we can easily enumerate p_1 . After obtaining p_1 , we can get the average inter-arrival time at station 1, i.e., $E(S_1)/p_1$, by applying Little's theorem. Since there are n jobs in the system, the average cycle time for each job, i.e., the time between two visits to station 1 by each job, should be

$$T(n) = n E(S_1) / p_1.$$

This is the result we want in stage one.

Now, let us come back to the original open system. Avi-Itzhak and Heyman propose to view the open computer system as an M/G/D queue, as shown in Figure 9, with arrival rate λ , D servers, and state-dependent service rate μ_n . If each job takes $\bar{a}$ cycles on the average, then clearly,

$$\mu_n = \begin{cases} n/\bar{a}T(n) & n = 1, \dots, D. \\ D/\bar{a}T(D) & n \geq D. \end{cases}$$

which is the rate jobs depart from the system.

Denoting the steady-state probability of having n jobs in the system by P_n , we can have the following set of balance equations:

$$\begin{aligned} r_0 P_0 &= \mu_1 P_1, \\ (r_n + \mu_n) P_n &= r_{n-1} P_{n-1} + \mu_{n+1} P_{n+1}, \quad n = 1, 2, \dots \end{aligned}$$

The solution to this set of equations is

$$\begin{aligned} P_n &= P_0 \prod_{i=0}^{n-1} (\lambda_i / \mu_{i+1}), \\ \sum_{n=0}^{\infty} P_n &= 1. \end{aligned}$$

If we assume $\lambda > \mu_D$, and $\lambda_0 = \lambda_1 = \dots = \lambda$, we get

$$P_n = \begin{cases} P_0 \lambda^n / \mu_1 \mu_2 \dots \mu_n, & n = 1, 2, \dots, D. \\ P_D (\lambda / \mu_D)^{n-D}. & n \geq D. \end{cases}$$

By Little's Theorem, we obtain the expected time a job spends in the whole system

$$E(T_S) = 1/\lambda \sum_{n=0}^{\infty} n P_n.$$

This is the final result we are looking for.

For our queueing model, the calculation in stage one is much simpler since we only consider two stations ($k=2$). Therefore, there are only n elements in D_n and one in D_{n-x_1} , i.e., $n=x_1$. The equation for p_1 then becomes:

$$p_1 = \left\{ \sum_{x_1=1}^{r_1-1} x_1 \frac{a_2^{n-x_1}}{\beta_1(x_1)\beta_2(n-x_1)} + r_1 \sum_{x_1=r_1}^n \frac{a_2^{n-x_1}}{\beta_1(x_1)\beta_2(n-x_1)} \right\} \cdot \sum_{x_1=1}^n \frac{a_2^{n-x_1}}{\beta_1(x_1)\beta_2(n-x_1)}.$$

If we assume monoprogramming, i.e., $n \leq r_1 = p$, the above equation can be further reduced to:

$$p_1 = \sum_{x_1=1}^n x_1 \frac{a_2^{n-x_1}}{x_1! \beta_2(n-x_1)} \cdot \sum_{x_1=1}^n \frac{a_2^{n-x_1}}{x_1! \beta_2(n-x_1)}.$$

The rest of the calculation remains the same.

The advantage of this method is, obviously, the ease of enumeration. However, there are several problems with this model. First, we do not know how accurate it is to approximate an open computer system by an M/G/D queue. Second, this model does not consider the effects of finite memory size and things like the scheduling algorithm. Third, and the biggest reason, we do not know the value of a_2 (the ratio of service rates).

Of course, we can figure out how a_2 affects the performance. However, we do not know how the system architecture, memory allocation strategy, and other factors affect the value of a_2 . An analytic formula is very

difficult to get. One way, perhaps the only feasible way, is to get this value by simulation. However, if we must simulate to find a_2 , we might as well use simulation from the beginning.

2.1.3 Konheim and Reiser's Method

Figure 10 shows the queueing model studied and solved by Konheim and Reiser [32]. It is a two-stage tandem queue with feedback and a finite intermediate waiting queue. The arrival of jobs is a Poisson process of rate λ , and service times in both stages are exponentially distributed with α and β , respectively. The principal characteristic of the service in this network is blocking. The first-stage server is blocked and ceases to offer service whenever M jobs are enqueued in the second stage. In other words, stage two cannot accumulate more than M jobs. Service resumes in the first stage when the number of jobs in the second stage falls to $M-1$. As usual, when a job leaves the second server, it will depart from the system with probability $1-\sigma$ or rejoin the first queue with probability σ .

This queueing model is, of course, quite different from our model shown in Figure 6. The only thing we are interested in is the method Konheim and Reiser use, which we can apply to our problem with a little modification.

In fact, the method they use is very basic, namely, using the state balance equations. For the queueing network shown in Figure 10, they come out with the following equation:

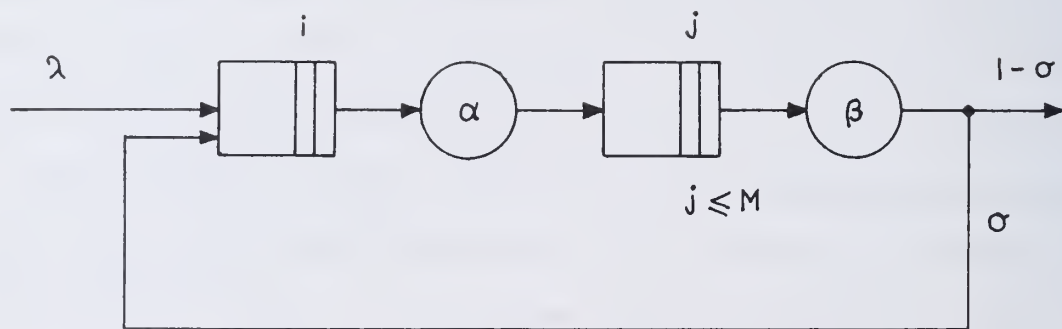


Figure 10. A Tandem Queue with Finite Waiting Room and Blocking.

$$\begin{aligned}
[\lambda + aI_{(i>0, j<M)} + \beta I_{(j>0)}] P_{i,j} = & \lambda I_{(i>0)} P_{i-1,j} \\
& + \beta a I_{(i>0, j<M)} P_{i-1,j+1} \\
& + \beta (1-\sigma) I_{(j<M)} P_{i,j+1} \\
& + a I_{(i>0)} P_{i+1,j-1}, \quad 0 \leq i \leq \infty, 0 \leq j \leq M.
\end{aligned}$$

$P_{i,j}$ is the steady-state probability that there being i jobs in the first stage and j jobs in the second stage. I_A is the indicator function of the condition A, which will be one if the condition holds and zero otherwise. The left hand side of the equation shows the possibilities that will cause the system to leave the states (i,j) , and the right side contains the possibilities that will cause the system enter this state.

Although the result Konheim and Reiser obtain from their analysis cannot be applied to our model, we certainly can use the method described above, except we have to increase the number of indices to three since our model has three stages.

Let us recall our model in Figure 6. Assume P_{ijk} is the steady-state probability that there are i jobs enqueued in the waiting queue, j jobs in the processor stage, and k jobs in the I/O stage, we can have the following balance equation which is similar to Konheim and Reiser's equation.

$$\begin{aligned}
[\lambda + f(j) + g(k)] P_{ijk} = & P_{i(j+1)(k-1)} f(j+1) \\
& + P_{i(j-1)(k+1)} g(k+1) \sigma \\
& + P_{(i-1)jk} \lambda P_r \{ \text{new arrival cannot enter} | \text{system} \\
& \quad \text{is in state } (i-1, j, k) \} \\
& + P_{i(j-1)k} \lambda P_r \{ \text{new arrival can enter} | \text{system is in} \\
& \quad \text{state } (i, j-1, k) \}
\end{aligned}$$

$$\begin{aligned}
& + P_{i,j}(k+1)g(k+1)(1-\sigma) P_r\{\text{no job can enter}|\text{system is} \\
& \quad \text{in state } (i,j,k+1)\} \\
& + P_{(i+1)(j-1)(k+1)}g(k+1)(1-\sigma) P_r\{\text{a job can enter}|\text{sys-} \\
& \quad \text{tem is in state } (i+1,j-1,k+1)\} \\
& \quad \text{for } i \geq 0, D \geq j+k \geq 0.
\end{aligned} \tag{4}$$

Where $f(j)$ is the total service rate of processors when j jobs are in the processor stage, and similar $g(k)$ is the total service rate of I/O devices when k jobs are in the I/O stage. We drop the indicator function from the equation. Of course, any term on the right hand side will vanish if it has a negative index.

The calculation of those conditional probabilities is a very interesting subject. It depends on several things, for example, the job scheduling algorithm, the memory allocation strategy, the total memory size (M), and the job size distribution. Of course, it also depends on the state (i,j,k) . Perhaps the best way to explain this is to give an example.

Let us look at the first conditional probability, namely, the probability that the new arrival cannot enter the memory, given that the system is in state $(i-1,j,k)$. If we assume a first-come-first-serve (FCFS) scheduling algorithm, then, trivially, this probability should be 1 when there is more than one job in the waiting queue, i.e., $i-1 \geq 1$ or $i \geq 2$. However, when $i=1$, i.e., the system is in state $(0,j,k)$, this probability will become the probability that this new job can enter the memory given there are already $j+k$ jobs in there. To answer this question, we must know the last three things we mentioned in the last paragraph. Let us assume we use distributed memory allocation (cf. Chapter 1), and the job size is normally distributed with mean μ and variance v^2 , then the probability will be:

$$\Phi\left(\frac{M-(j+k)\mu}{\sqrt{j+k} \ v}\right) - \Phi\left(\frac{M-(j+k+1)\mu}{\sqrt{j+k+1} \ v}\right)$$

where $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt.$

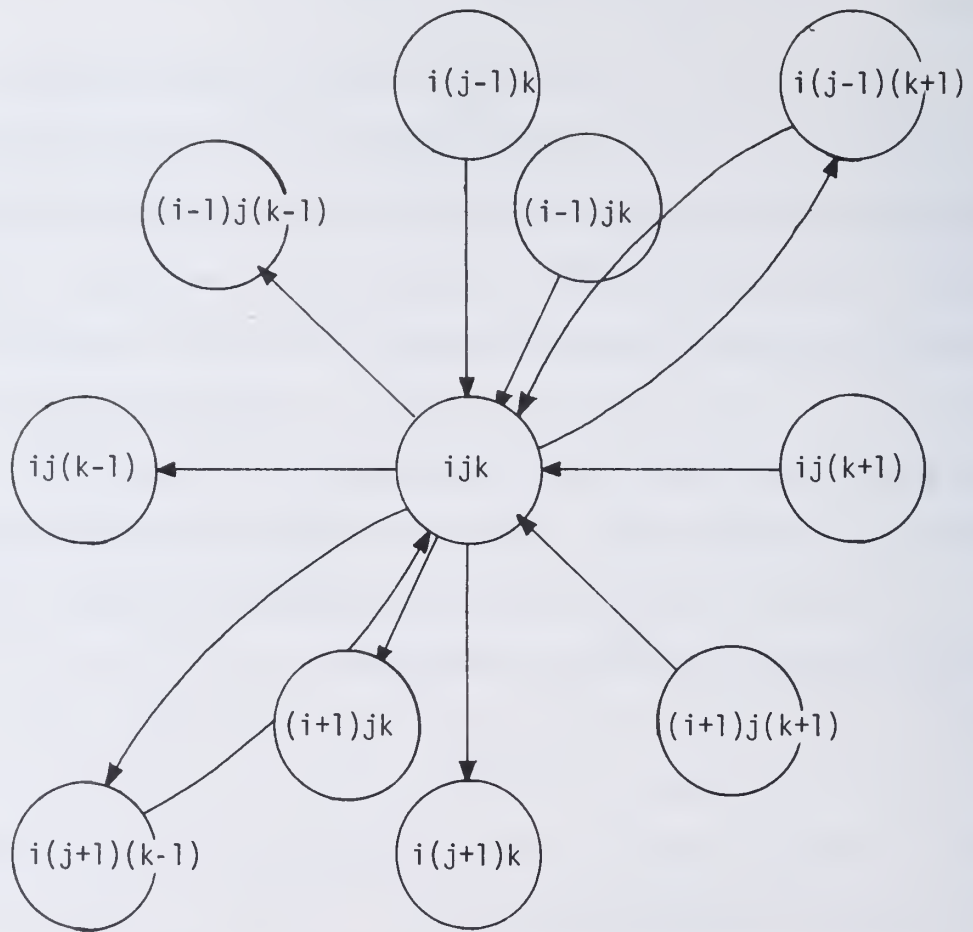
As we can see, all the terms on the right hand side might have quite different forms from equation to equation, since they depend heavily on the state and other factors. In general, it is impossible to solve equation (4) analytically by using a transformation technique. This is the same problem Konheim and Reiser encountered, although their equation is much simpler than ours. If we only want to solve the state probability, then we can use a numerical method Konheim and Reiser use for their model. The method is very straightforward. However, it requires two huge arrays.

If we let $P = [P_{ijk}]$ be the state probability "vector," then we can rewrite equation (4) in the following form:

$$P = PA$$

where A is the state transition matrix. Each entry in A can be enumerated by the method we just described.

Figure 11 is a three dimensional representation of equation (4). We can see equation (4) forms an irreducible Markov-chain since every state can be reached by any other state. For example, from state $(i, j, k-1)$ we can go to state (i, j, k) via state $(i, j-1, k)$. Only the direct transitions to and from state (i, j, k) have been shown in Figure 11. The number attached to each arc represents the corresponding state transition on the right hand side of equation (4).



* Only transitions to and from state (i,j,k) have been shown

Figure 11. The Irreducible Markov Chain.

After getting the state transition matrix A , we can write a simple program to calculate the state probabilities. Here is an algorithm we can follow.

Step 1. Initialize P ; set $P_{000}=1$ and $P_{ijk}=0$ for all other elements. Also, $n=0$.

Step 2. $T \leftarrow A * P$, $n = n+1$.

Step 3. Is $n \leq \text{Limit}$? If not, stop, and check all parameters.

Step 4. Is $T \approx P$? If not, $P \leftarrow T$ and go back to Step 2.

Step 5. (Obtain steady-state probabilities) Calculate everything we want.

If the arrival rate λ is small enough, the process will converge eventually and we will obtain the steady-state probabilities. We can then calculate the average number of jobs in the system ($\bar{N}$), and the average turnaround time ($\bar{W}$) by using Little's Theorem. Let $n = i+j+k$, then

$$\bar{N} = \sum_{n=0}^{\infty} n P_{ijk},$$

$$\bar{W} = \bar{N}/\lambda.$$

Although the calculation of those conditional probabilities might be difficult sometimes, it indeed provides a great capability to measure a lot of systems. For example, instead of FCFS we can implement last-come-first-serve (LCFS), look-ahead (LA), or any scheduling algorithm we can formulate. Later in the next section, we will introduce a recursive method given by Chandy, et al [33], which is very useful on this subject.

One other advantage of this method is that it includes the effects of finite memory size and the limited number of jobs allowed in the memory.

This is what Avi-Itzhak and Heyman's method cannot supply.

However, one problem still has not been solved yet, i.e., what the service rates of processors and I/O devices are. This is why we did not talk about the functions $f(j)$ and $g(k)$ contained in equation (4), because it is rather difficult to formulate them to cover the effects of memory allocation and memory interference, especially in a partial connection architecture. Again, we reject this method, since the service rates are very crucial in getting correct answers.

2.1.4 Brown, Browne and Chandy's Method

Brown, Browne and Chandy's model [33] is designed for the analysis of a computer system with a nonpaged memory executing a multiaccess interactive workload. The actual system is the dual CDC 6400/6600 computer complex at the University of Texas at Austin. Their main purpose is to study the effects of the finite size of executable memory, job workload characteristics, system overheads, and swap times on the system performance. Therefore, they do share some common interests with us. The only difference is that we are also interested in the effect of hardware architecture. In other words, Brown, Browne and Chandy are trying to improve a currently existing system, but we are trying to design a better system for the future.

Figure 12 shows a network representation of the overall system, to be modeled. When a job is at the user terminal (physically the job is in the extended core), it will suffer a "thinking time" delay which corresponds to the time period from the last response until the user submits the next command. Then, the job will be queued for swap-in. This is represented by Swap Delay. When the job is allowed to enter the main memory, it will first need some amount of time to be swapped from ECS to the main

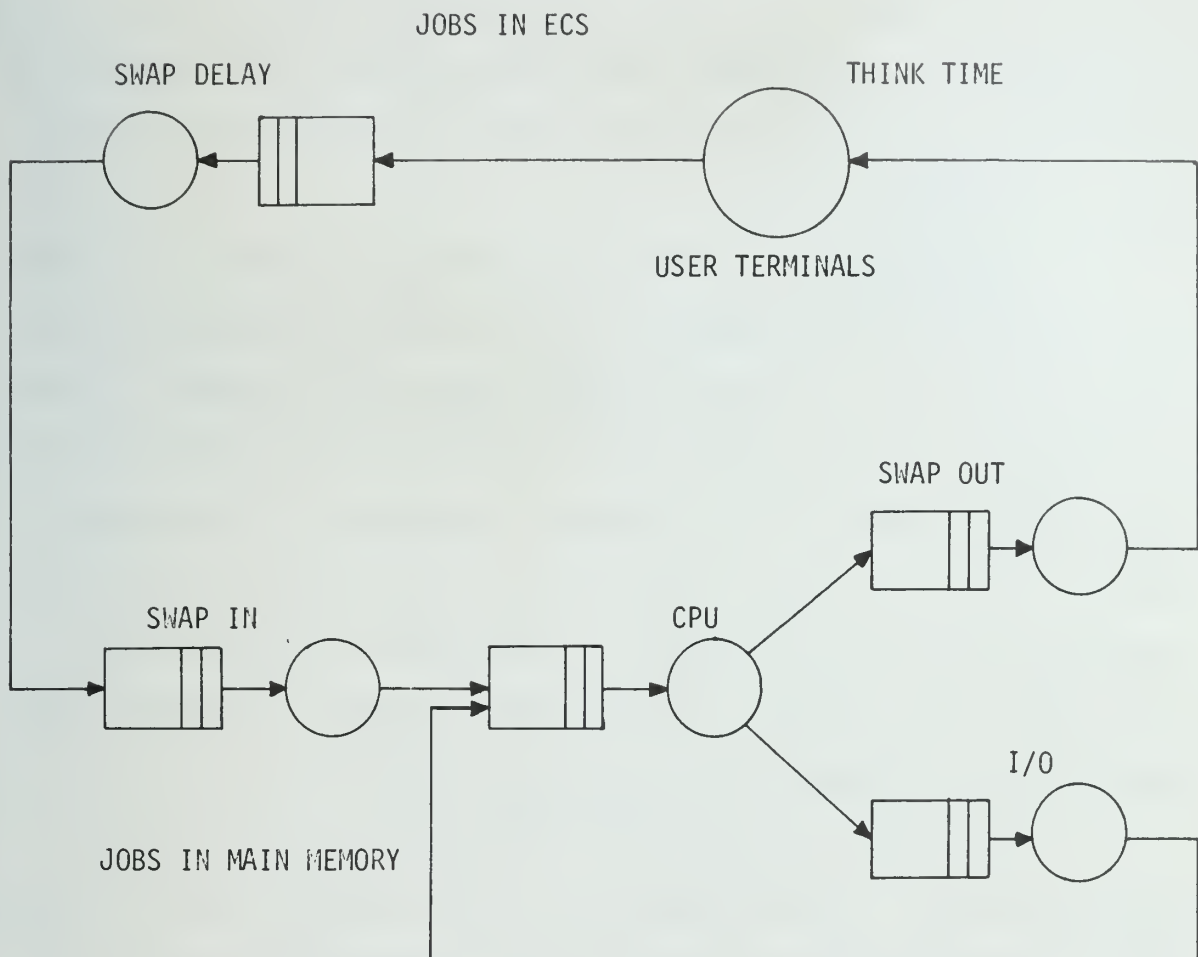


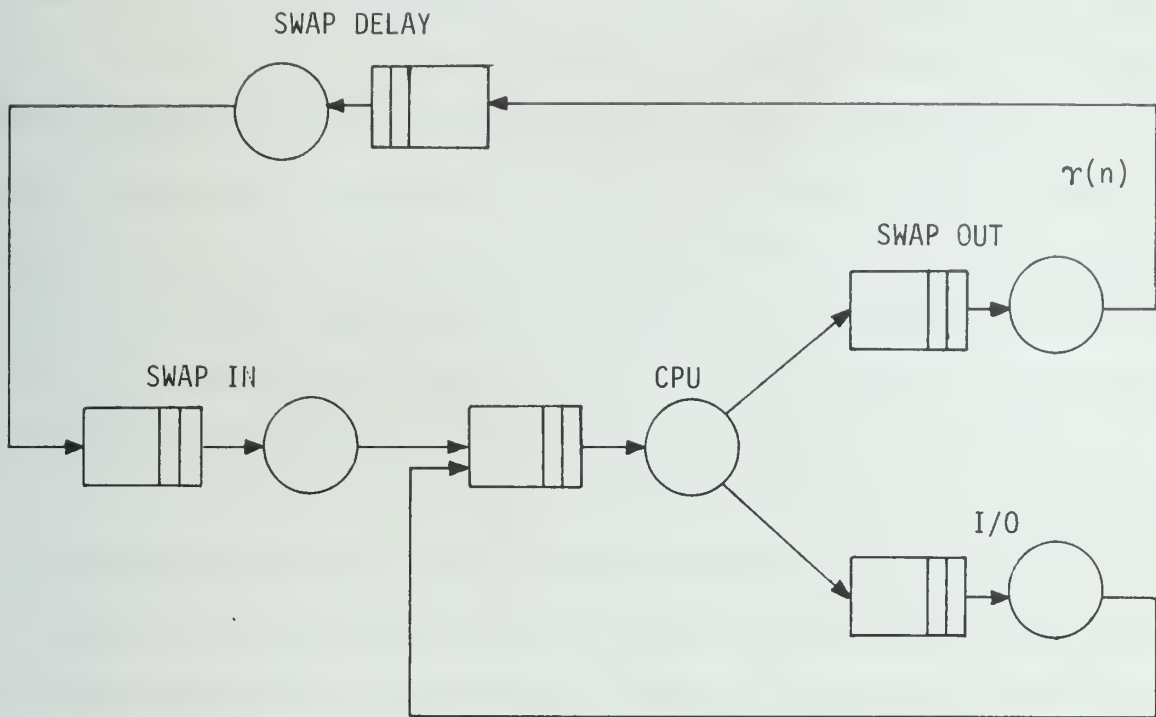
Figure 12. Overall System to be Modeled.

memory, then it will queue for CPU service. After the CPU processing, the job will either go to the I/O stage or be ready to be swapped out. If it goes to the I/O stage, it will rejoin the CPU queue for another service after the I/O operation. If the job is ready to respond to the user terminal, it will be swapped out of the main memory to ECS, and the whole cycle begins again.

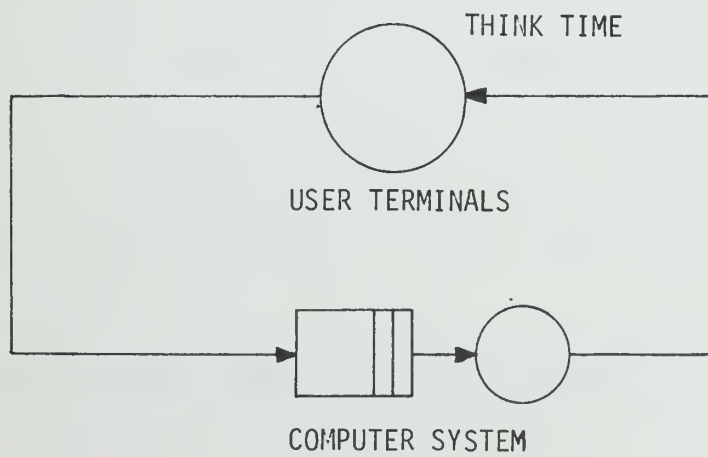
If the thinking time, the swapping delays, and the service times are all exponentially distributed, the whole system is a Poisson queueing network and we can apply Gordon and Newell's result [36]. However, due to the finiteness of the main memory size, a job might get blocked in the swap delay stage even though it is at the head of the queue. In other words, because of insufficient memory space, the job might suffer an unpredictable queueing delay before being swapped in. This means the system will not behave exactly like a Poisson queueing network any more. Therefore, some other technique is needed to account for this effect.

A hierarchical decomposition technique [37] is then used to analyze the system. They decompose the system of Figure 12 into two systems shown in Figure 13, and use the following four computational steps to get the result.

Figure 13(a) is called the device model. It is identical to the overall system of Figure 12, except the think times are assumed to be zero in the device model. Now, the device model is analyzed assuming there are n jobs in the model. All these n jobs are also assumed to have no problem of getting into the main memory. In other words, this is equivalent to a closed Poisson queueing network with n jobs in it and no memory constraint. Hence, we can easily determine $\gamma(n)$, the expected number of jobs swapped out



(a) Device Model



(b) Overall Model

Figure 13. Component Models of the Overall System of Figure 12.

per unit of time given there are n jobs in the system, or the throughput out of the swap-out queue. The analytic solution can be obtained by applying Jackson's or Gordon and Newell's equation.

In the second step, we will compute $h(n|m)$ which is assumed to be the conditional probability that n jobs have been allocated memory given that there are m ready jobs. By a ready job, we mean a job is either waiting to be swapped in, or already in the main memory. When a job gets swapped out, it leaves the ready state and enters the think state. Brown, Browne and Chandy propose the following method to compute this quantity recursively.

Let us assume all the ready jobs form a single first-come-first-serve queue. A job will join the end of this queue whenever it becomes ready, and will be removed from the queue whenever it goes back to the think state. Those jobs already in the main memory tend to cluster in the front part of the queue. Now, let us go through the ready queue and compute $g(n,y|\ell)$, for all $n = 0,1,\dots,m$ and $y = 0,1,\dots,c$, where c is the total memory size. $g(n,y|\ell)$ is the conditional probability that n jobs have been allocated y units of memory, given that the first ℓ ready jobs have been considered. Obviously, we have the following simple facts:

$$g(n,y|\ell) = \begin{cases} 1, & \text{for } n=0, y=0, \ell=0, \\ 0, & \text{for } n=0, y>0, \text{ all } \ell, \\ 0, & \text{for } n>\ell, \text{ all } y. \end{cases}$$

Then, $g(n,y|\ell)$ can be computed from the following recursive equation:

$$\begin{aligned}
 g(n, y | \ell) &= g(n, y | \ell - 1) [1 - P(c - y)] \\
 &+ \sum_{x=0}^y g(n-1, x | \ell - 1) p(y-x), \quad \text{for } \ell = 1, 2, \dots, m.
 \end{aligned}$$

Where $p(x)$ is the probability that a job will be of size x , and $P(x)$ is the cumulative probability, i.e., $P(x) = \sum_{y=0}^x p(y)$. The proof of the above equation is very simple and can be found in [33]. After getting all these conditional probabilities, we can calculate $h(n|m)$ by summing over all values of y , i.e., $h(n|m) = \sum_{y=0}^c g(n, y | m)$.

In the third step, we will consider the queueing network shown in Figure 13(b), which is called the overall model. This model consists of just user terminals and a single queue/server called the computer system. All the servers in the device model are coalesced into a single server. The number of jobs in this model is the total number of jobs in the system, i.e., M . The ready jobs are those in the computer system. Let $q(m)$ be the expected number of jobs serviced by the computer per unit of time given that there are m jobs ready. It can be computed by

$$q(m) = \sum_{n=1}^m \gamma(n) h(n|m), \quad \text{for } 1 \leq m \leq M.$$

Therefore, what we have now is the queue-length-dependent service rate of $g(m)$ when there are m jobs in the computer queue. We can then apply the balance equation technique to solve the state probabilities, and calculate the system throughput.

Basically, this method uses the same technique as the first method by Avi-Itzhak and Heyman. Both methods want to find out a state-dependent service rate first and use the classical balance equation method to solve

the rest of the problem. However, this method is more powerful and accurate since it considers several things the first method does not include.

Unfortunately, like the other two methods, this method still does not include the effects of system architecture and resource contention. In addition, the calculation of $g(n,y|l)$ is only limited to very simple scheduling algorithms. Since we are interested in the effects of all these factors, we have decided not to use an analytic approach. Instead, we will use simulation which allows us to consider as many parameters as we want.

In the next section, we will talk about how we do the simulation and discuss some problems associated with the simulator. We will also give some definitions of the parameters we are measuring which will be very useful in our discussion in the next chapter.

2.2 Simulation

In the last three sections, we talked about some analytic tools for studying computer performance. Although we devoted quite a number of pages to these methods, what we really wanted to do was to show their limitations and to explain why we cannot use them for our work.

In Chapter 1, we discussed several problems we are interested in: we want to compare several memory allocation schemes; we want to study the effect of scheduling algorithms; we want to compare partial connection with full connection; we want to see whether we should use multiprogramming or monoprogramming; and so on. All these make our system extremely complicated, and none of those analytic models can cover all the problems. Hence, the simulation technique must be used to meet all our requirements.

Although simulation is a very expensive method, it can handle any

arbitrarily complicated model. We can increase or decrease the system complexity at our will. It is this ability to cope with reality that makes simulation more useful than queueing analysis.

Before we describe our simulator, we would like to discuss a problem, namely, the memory bandwidth problem. By memory bandwidth, we mean the number of words we can access in the main memory in a unit of time. Usually, bandwidth will be measured in number of words per memory cycle. In other words, the memory bandwidth represents the information flow rate in or out of the main memory. Since most of the processor operations are related to the memory, e.g., the instruction and the operand fetches, memory bandwidth significantly affects the system throughput. The higher the bandwidth is, the faster the processors operate.

Hence, in order to determine how fast the system operates, we must know how much memory bandwidth we can get from the system. As we will see later, this is what we use to advance the virtual clock in our simulator.

In the next section, we will first derive a simple bandwidth equation. Then, we will show a general equation we use in the simulator. This general equation can be used to handle different kinds of memory allocation and different types of system architecture. We can also put in some parameters, e.g., the memory-processor speed ratio, in order to take care of things that might exist in a real computer system.

2.2.1 Memory Bandwidth Problem

We just defined the memory bandwidth to be the average number of words we can access during one memory cycle. From the memory's point of view, it is the average number of busy memory modules per unit of time. Of course, this quantity depends on a lot of factors, for example, how many references

a processor will make in one memory cycle; the inter-relationship or pattern of these references; how many processors are accessing the memory; how do they interface with each other; etc. In general, this is not a simple problem to solve. However, if we can make some reasonable assumptions, then we might be able to come out with some closed form solutions.

Let us start with a simple problem. Assuming there are m identical memory modules operating in parallel, and a processor is generating s randomly distributed references to these memory modules per memory cycle, what is the memory bandwidth of this system? For example, the processor is s times faster than the memory and it is accessing s items at random addresses. Or equivalently, there are s (usually we use p , so $s=p$ in this case) independent processors and each one is only making one memory reference per memory cycle. This is an interesting combinatorial problem whose answer is given by the following theorem [38].

Theorem: Given m identical memory modules operating in parallel, if we generate s randomly distributed references, then the average number of busy memory modules (bandwidth) will be:

$$B_w = \sum_{k=1}^t k \frac{k! S(s, k) \binom{m}{k}}{m^s}$$

where $t = \min(m, s)$ and $S(s, k)$ is the Stirling number of the second kind.

We prove in [39] that this equation can be reduced to a very simple closed form, that is:

$$B_w = m[1 - (1 - 1/m)^s]. \quad (5)$$

If we are interested in asymptotic behavior, then the above result can be transformed into the following linear form as we keep the ratio of s and m fixed and let m and s go to infinity.

$$B_w = m[1 - 1/e^r], \quad \text{where } r=s/m.$$

What this result implies is that, when we double the number of processors and memory modules, the memory bandwidth we will get is also doubled, provided each processor is generating one independent reference per memory cycle. This contradicts what people have always called the biggest disadvantage of a multiprocessor, namely, doubling the cost will not double the performance. The most famous result is, of course, the square root equation proposed by Hellerman [40], which says that the memory bandwidth of an interleaved memory system only grows as the square root of the number of memory modules. So, when we double the modules, the memory bandwidth only increases by roughly 40%. Apparently, this result is too conservative.

In fact, the result in equation (5) can be obtained in another way. Let us look at one specific memory module. Since each processor (or reference) will access this module with probability $1/m$, the probability it will not be accessed by a processor is $1 - 1/m$. Since all processors are independent, the probability that none of them will reference this module will be $(1 - 1/m)^S$. Therefore, the probability that at least one processor will reference this module is $1 - (1 - 1/m)^S$, which is the probability that this memory will be busy. Summing over all the memory modules, we will get equation (5) as the average number of busy modules, or the memory bandwidth. This method is very useful, and later we use it to get our general bandwidth

equation.

Now, let us go back to the first problem. It is generally acknowledged that references generated by a processor should not be considered as randomly distributed. Instead, there should be some kind of relationship between two references. Hence, Burnett and Coffman [41] introduced the effects of seriality. They assume that the probability of the next reference addressing the next module in sequence (modulo m) will be α , and the probability of addressing any other module will be β , where $\beta = (1-\alpha)/(m-1)$. Or formally, let r_i be the module number of i^{th} reference, then

$$\begin{aligned} P_r\{r_{i+1} = (r_i + 1) \bmod m\} &= \alpha, \\ P_r\{r_{i+1} \neq (r_i + 1) \bmod m\} &= \beta, \text{ for } i=1,2,\dots,s-1. \end{aligned}$$

Where s is the number of references generated per memory cycle. Then, the memory bandwidth is given by the following theorem [30,42].

Theorem: Assume the processor generates s memory references per memory cycle. If the next reference in line will address the next module in sequence (modulo m) with probability α and any other module out of sequence with probability $\beta = (1-\alpha)/(m-1)$, then the memory bandwidth will be:

$$B_w = \sum_{k=1}^t \sum_{j=0}^{k-1} \alpha^j \beta^{k-1-j} C_m(j,k),$$

where $t = \min(m,s)$, and

$$C_m(j,k) = \binom{k-1}{j} \sum_{n=0}^{k-j-1} (-1)^n \binom{k-j-1}{n} (m-j-n-1)_{k-j-n-1}.$$

If we plot B_w against a , we can see the bandwidth grows exponentially as a increases. This implies that, if a program has a high seriality a and if addresses are distributed across the memory modules, then we can get very high bandwidth out of the memory.

However, if there is more than one processor the problem becomes very complicated, since we should also consider the interference between processors. The solution for one processor is already very messy, it will be even more difficult to use the same kind of approach. Therefore, we do need a new technique for finding out the memory bandwidth.

Recall the probability approach we just described to derive equation (5). If we can figure out the probability that a certain module will be busy, then we can sum all these quantities together and obtain the total memory bandwidth. This is a very basic and useful idea of getting memory bandwidth. Using this idea, we can write down the following general bandwidth equation:

$$B_w = \sum_{j=1}^m [1 - \prod_{i=1}^p (1 - p_{ij})], \quad (6)$$

where m , p are the numbers of memory modules and processors, and p_{ij} is the probability that the i^{th} processor will reference the j^{th} module. Of course, the only problem is to find out all the p_{ij} 's. For example, let all $p_{ij} = 1/m$, then equation (6) is reduced to equation (5).

Let $q_{ij} = 1 - p_{ij}$ be the probability that the i^{th} processor will not reference the j^{th} module. The above equation can be rewritten as:

$$B_w = m - \sum_{j=1}^m \prod_{i=1}^p q_{ij}.$$

Sometimes it is easier to get q_{ij} then p_{ij} .

Now we can solve the multiprocessor bandwidth equation problem mentioned above. We state the problem and the solution in the following theorem. The only thing we have to do is to figure out p_{ij} 's and substitute into equation (6).

Theorem: Assume we have p processors referencing m memory modules. Each processor generates s references per memory cycle and they have the seriality relationship stated in the previous theorem. Let $p_{ik}^{(\ell)}$ be the probability that the i^{th} processor will generate a k in the ℓ^{th} position and no j occurs before, i.e., the probability of the event shown in Figure 14, then the bandwidth equation for $s > 1$ will be:

$$B_w = m - \sum_{j=1}^m \sum_{i=1}^p q_{ij}^{(s)}$$

where

$$q_{ij}^{(s)} = (1 - p_{ij}^{(s-1)}) \left\{ 1 - \left[\frac{p_{i(j-1)}^{(s-1)}}{1 - p_{ij}^{(s-1)}} a + \left(1 - \frac{p_{i(j-1)}^{(s-1)}}{1 - p_{ij}^{(s-1)}} \right) \frac{1-a}{m-1} \right] \right\}$$

The proof of this theorem and the calculation of $p_{ij}^{(s)}$ are given in Appendix A. Another derivation by using Markov chains is also given there.

This theorem shows the usefulness of equation (6). We can use it for deriving the memory bandwidth of a great variety of systems. We will also use equation (6) in our simulator. However, the real meaning of a is very vague. Every program has a different value of a . Unless we do a thorough analysis of program traces, we are in no position to say what the

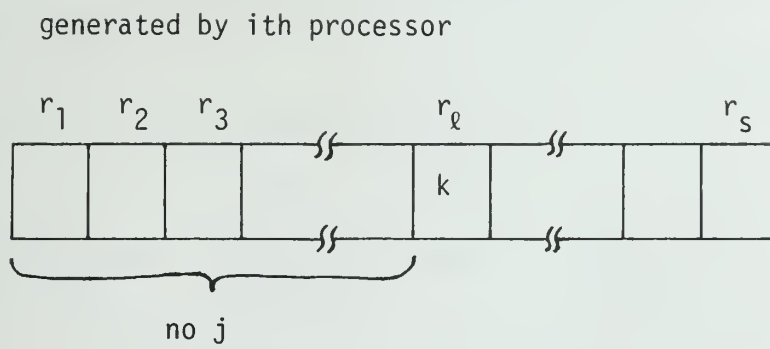


Figure 14. The Event for Computing $p_{ik}^{(l)}$.

α value for program should be. We cannot afford doing this in our study. Besides, when s is small, say 2 or 3, moderate change of α really does not make too much difference in the result. Therefore, we prefer not to use the above recursive solution. Instead we just assume no seriality between references, i.e.,

$$q_{ij}^{(s)} = (1 - p_{ij})^s$$

A good discussion of memory bandwidth can be found in [43] or [39].

2.2.2 The Simulator

Our simulator uses the so-called event-driven technique, that is, the whole simulation process is driven by a sequence of event times. An event time is the time that an event occurs, which could be the arrival of a job, the completion of a processing period or I/O operation, the departure of a job, etc. The virtual clock is advanced from the current time to the next event time. Every time we advance the clock to a new event time, we will calculate all the statistics we want between this new event time and the previous event time, and update the system status. Then, we will use this new status information to figure out the time that the next event will occur. This process will keep going until a certain number of jobs has been simulated. In a simulation where the timing is the most important statistic, the event-driven technique is the most convenient and useful tool.

Figure 15 is the overall structure of the simulator, and Figure 16 is the flow chart of the main program. Obviously, the most important part is how to generate the next event time (the dotted box in Figure 16).

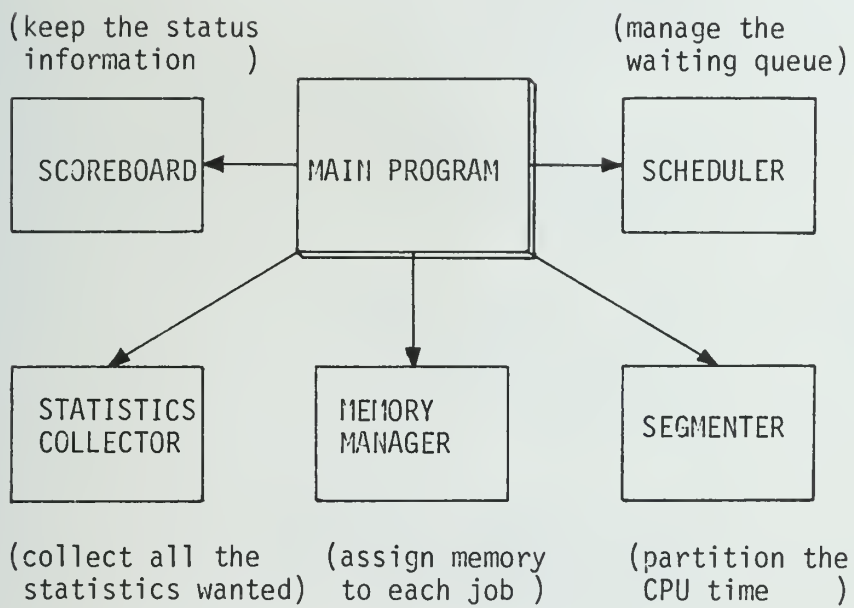


Figure 15. Overall Structure of Our Simulator.

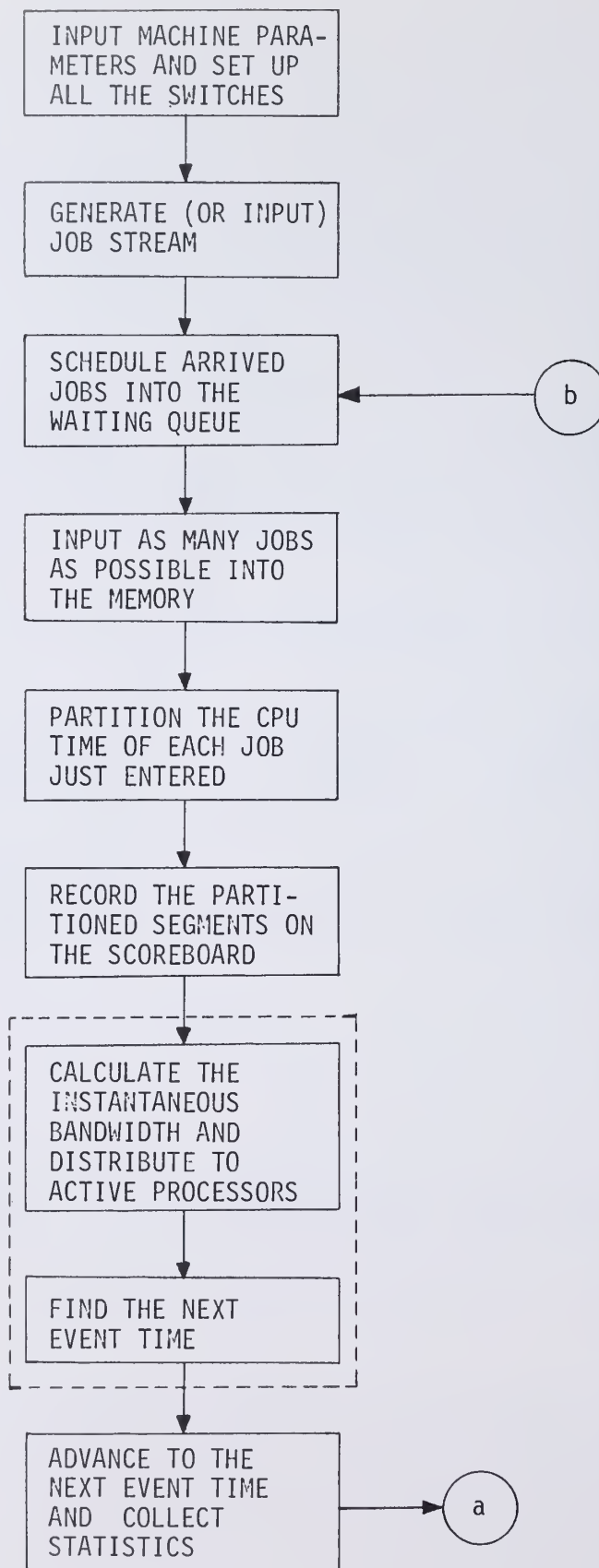


Figure 16. Flowchart of the Simulator.

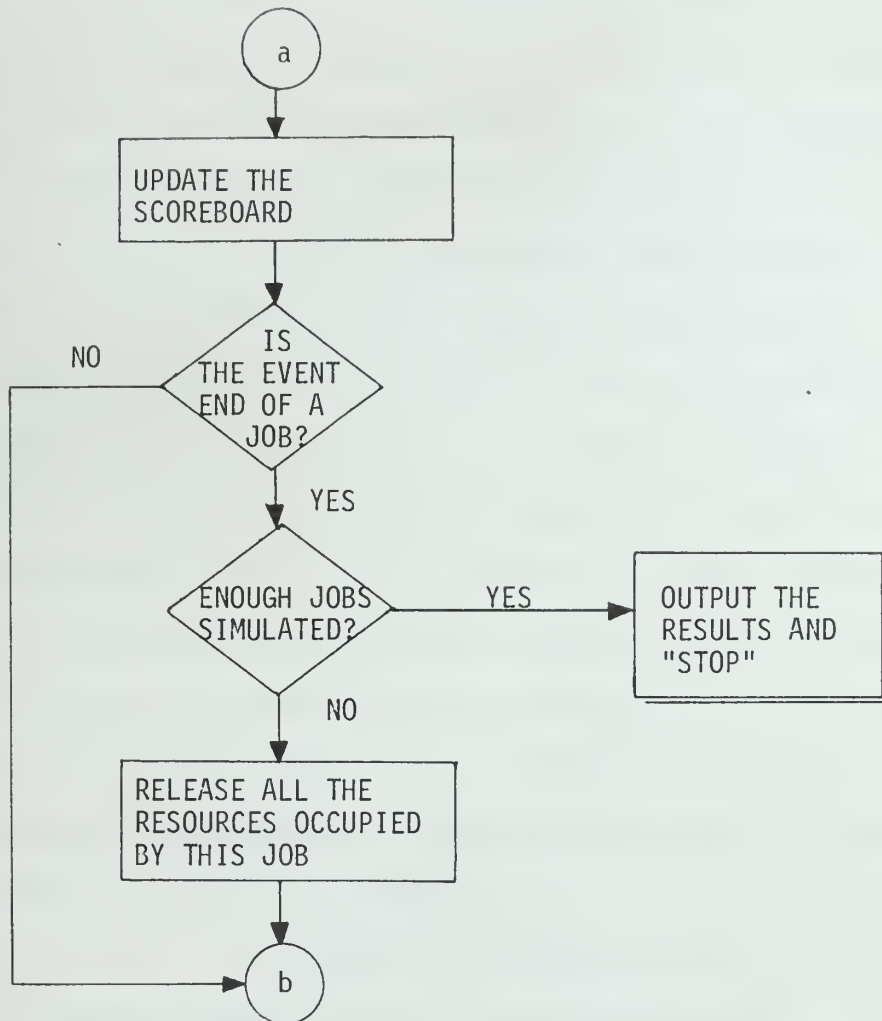


Figure 16. Flowchart of the Simulator. (Continued)

The input to the simulator is a sequence of jobs. Each job is a four-tuple that consists of the arrival time of this job, as well as the CPU time required (in milliseconds), the number of I/O requests, and the memory space (in K bytes) required by this job. Usually, people call this kind of information a "workload." We use two kinds of workload in our analysis, namely, the real workload and the artificial workload. The real workload is obtained from the System Management File (SMF) in our IBM 360/75 system. The SMF routines store on tape a complete record of the processing information of all the jobs run on the 360/75 every day. We pull off all the above information from the SMF tapes to constitute the input workload of our simulator. On the other hand, we can use random number generators to generate an artificial job stream. The means and variances of the job parameters can be obtained by analyzing the real data we got from the SMF tapes, and these can be used by the random number generators to produce fake jobs. The real workload can reflect what really happened in a computer system, but the artificial workload is easier to modify. We will use both and compare their results.

When a job "arrives," that is, when the content of the virtual clock is equal to or greater than the job arrival time, it will be placed into the waiting queue according to some scheduling algorithm. The scheduling algorithm will greatly influence the system performance, especially the average turnaround time. In the next chapter, we will compare some non-preemptive scheduling algorithms.

The jobs in the waiting queue are then considered for entering the system according to their ordering. If the memory has enough room for the job under consideration, this job will join the system and start getting

service. Of course, in a monoprogramming system this job also should have a free processor assigned to it. In the job scanning, we usually allow a certain distance of look-ahead. This means, if a job cannot be selected for service, we are allowed to go down the line and look at the next job. This scheme might improve the performance. It is particularly true for short look-ahead distance. But for long look-ahead distance, we might get some negative result, since allowing large look-ahead tends to reduce the effect of the scheduling algorithm. Our result in the next chapter indeed shows this phenomenon.

One negative side of the look-ahead scheme is that a job requiring large memory space might get blocked all the time since smaller jobs might sneak in and never leave enough space for this big job. In order to avoid this problem, we adopt the following strategy. When a job first joins the waiting queue, we will attach a count to this job. Later on, we will reduce this count by one every time we scan this job. When this count becomes zero and has not been served, we will stop looking ahead and force the system to accept this job eventually. In the 360/75 system, people use a similar method to accomplish this, namely, by gradually reducing the magic number attached to a job. Of course, the 360/75 also uses a more complicated method, viz. adjusting the job initiators to give large jobs more chance to get into the memory.

When a job gets into the memory, we first partition its CPU time into as many pieces as the number of I/O requests required by this job in the following way. Let us assume the job requires i I/O requests. We generate i exponentially distributed random numbers in $(0,1)$, sum them together, divide the total CPU time by this sum to get a proportionality constant, and then multiply each original random number by this constant to become the

length of each piece. It is easy to show the sum of these normalized pieces is equal to the total CPU time.

The reason we are doing this is that we do not know the length of the time period between two I/O requests. Although the SMF tapes do contain this information, it is difficult and time-consuming to get it from the tapes.

During the simulation process, a job will go to the I/O stage after one segment of CPU time has been served. It will then do some I/O and return to process the next segment. Each I/O operation will be assumed to be nominally 42 ms long, which is of the same order as a disk operation on a 2314 disk unit. Of course, this parameter can be changed.

As we said before, the virtual clock of the simulator advances from an event time to the next event time. The next event time depends on how fast the system operates, and this in turn depends on the memory bandwidth. The faster we can get data out of the memory, the faster the system can operate.

The memory bandwidth is a function of several variables, for example, the processor-memory speed ratio, the number of active processors, the number of memory modules, and the memory allocation scheme. In the last section, we showed a general equation which will be used in our simulator. This equation is used in the dotted box of Figure 16. However, one thing we still have not mentioned, i.e., p_{ij} in equation (6). p_{ij} depends on what percentage of the program resides in this module and how we allocate memory to the program. In our simulator, we will assume the program is interleaved horizontally, i.e., across the memory modules. So, it is fair to assume p_{ij} is the fraction of the program residing in a certain module.

In other words, if a program is distributed in four memory modules, then all four p_{ij} 's will be assumed to be $1/4$. Of course, some people might argue about this, but we think it is a reasonable assumption.

At any time instant, how fast those processor can run is determined by the "instantaneous" bandwidth. By instantaneous bandwidth, we mean the memory bandwidth of the current state, or before the state change. This is because the memory bandwidth is state-dependent, i.e., depends on how many processors are running.

Once the instantaneous bandwidth has been figured out, we will distribute it to all the "active" processors. The share each processor will get is proportional to the contribution it makes toward the total bandwidth. This partial bandwidth can be viewed as the processing power the processor uses to execute a job. We can then figure out the time that the next event will occur and the amount of work each processor has done between two event times. In general, the instantaneous bandwidths of two intervals will be different, because the number of active processors might be different.

Now, let us give a simple example to show how to calculate the memory bandwidth and compute the next event time. Assume we have a distributed system with eight memory modules ($m=8$) and a speed ratio of two ($s=2$). Also, assume we have three jobs in execution that require 10.5, 13.0. and 20.8 miliseconds CPU time until their next I/O operations, and the current time T equals 2145.32 (all these figures are arbitrarily chosen). Since we are using the distributed memory allocation scheme, all p_{ij} 's will be equal to $1/8$.

First, we have to figure out the total instantaneous bandwidth.

This can be obtained by using the following formula:

$$B_w = 8 - \sum_{j=1}^8 \sum_{i=1}^3 q_{ij}^2,$$

where $q_{ij} = 1 - p_{ij}$. Since $q_{ij} = 7/8$, we can get $B_w = 4.41$. If we assume another memory allocation scheme, e.g., the partitioned or mixed scheme, the calculation of the total instantaneous bandwidth is very similar, although it will be a little bit more complicated. In the next chapter, we will show the calculation for a partitioned system. Since we are using the distributed scheme, the total bandwidth will be equally distributed to all the jobs. Therefore, each processor will get a bandwidth of 1.47 to execute the job.

Now, we can compute the next event time by using all the information provided above. Apparently, the job with 10.5 milliseconds of work will be done and go to the I/O stage first. In other words, the next event time is the time instant this job stops processing and issues an I/O request. The other two jobs will still be in execution by their processors by then, since all the jobs get the same amount of bandwidth and will progress at the same time. So, the problem is to find out how long it will take to finish 10.5 milliseconds of work, given the processor has a memory bandwidth of 1.47. We can solve this by manipulating the dimensions.

Let us assume w to be the average number of memory references a processor will issue every milliseconds and w' to be the number of memory cycles per milliseconds. The dimension of memory bandwidth is the number of memory references per memory cycle. So, the time to finish 10.5 ms of work is:

$$\Delta T = \frac{10.5 \text{ ms} \times w \text{ mr/ms}}{1.47 \text{ mr/mc} \times w' \text{ mc/ms}} = 7.14 \frac{w}{w'} \text{ ms.}$$

If we assume a processor issues a memory reference every processor cycle, then w/w' is equal to the memory-processor speed ratio s . (In fact, we define processor cycle time to be the average period of time a processor will take to issue a memory reference.) Hence, ΔT is equal to $7.14 \times s = 14.28 \text{ ms}$. The next event time will be $T + \Delta T = 2145.32 + 14.28 = 2159.6 \text{ ms}$. In other words, the first job will stop processing and issue an I/O request time at time instant 2159.5. Since we assume each I/O operation will take a constant amount of time, we can know the time this job will finish the I/O operation if it gets the I/O device it wants.

Of course, we must assume that nothing else will happen between the current time T and $T + \Delta T$. For example, if a fourth job finishes an I/O transaction and resumes processing between these two time instants, the next time will be the time instant that this fourth job resumes processing. Hence, instead of computing ΔT , we have to figure out how much work will have been done between the current time and the time instant the above event occurs. We then have to subtract this from the processing time of each job and repeat the whole thing with four jobs.

We can see that the principle behind our simulator is quite simple. However, it is a very useful tool, and we use it to generate all our results. In the next chapter, we will show some interesting results we have obtained. Before we do that, we will define some of the measurements which will be used in the later discussion.

2.2.3 Definitions of System Measurements

In order to give a more succinct presentation of the simulation results, we will use the following definitions very often in the next chapter. As always, p , m , and r denote the numbers of processors, memory modules, and I/O devices in the system. The memory-processor speed ratio, i.e., the ratio of memory cycle time and processor cycle time, is denoted by s .

T_a will be used to denote the average turnaround time, i.e., the average amount of time a job will spend in the system. T_a can be broken into two parts, namely, the average queueing time $\bar{q}$ and the average service time $\bar{e}$. In other words, $T_a = \bar{q} + \bar{e}$. $\bar{q}$ is the average amount of time a job has to spend in the outside waiting queue, or the time period from the moment the job arrives to the moment it enters the memory. In a multiprogrammed system, $\bar{q}$ is usually caused by insufficient memory space. In a monoprogrammed system, this could also be caused by lack of a free processor. Sometimes, we might put a superscript on $\bar{q}$ to denote the queueing time which occurs somewhere else. For example, $\bar{q}^{i0}$ represents the queueing time which occurs in the I/O queue. On the other hand, $\bar{e}$ denotes the average amount of time a job will spend in the memory, which can further be broken into processing time, I/O time, and some possible delays due to queueing for resources, e.g., $\bar{q}^{i0}$. We gave a graphical representation of these parameters earlier in Figure 7.

$\bar{n}$ will denote the average number of jobs in the whole system. Again, a superscript will be used to denote which part of the system we are talking about. For example, $\bar{n}^q$ represents the average number of jobs in the outside waiting queue.

B_w is used to represent the memory bandwidth. A superscript will be used to indicate the memory allocation scheme we use. So, B_w^d , B_w^m , and B_w^p will denote the memory bandwidths for distributed, mixed, and partitioned systems respectively.

U_m and U_p will denote the utilizations of memory and processors, U_m is the average fraction of the memory which will be occupied by jobs. We will explain later that there are two kinds of memory utilization in a partitioned system due to an unusual way of allocating memory. U_p , on the other hand, is the fraction of time a processor is busy executing a job.

Chapter 3 EXPERIMENTAL RESULTS

3.1 Results for Software Related Questions

We roughly described some interesting design problems in Section 1.4. In this chapter, we are going to present a lot of simulation results to answer these problems. Each problem is affected by a number of variables, and we will include the effects of as many of these variables as possible. Basically, we will follow the same order as that in Chapter 1. Therefore, we will start with software related questions.

Before we proceed, a word of caution is in order. In sections 3.1.2 and 3.1.3 which deal with monoprogramming versus multiprogramming and the memory allocation schemes, the reader will quickly come to the conclusion that multiprogramming and the distributed memory allocation scheme are superior in terms of performance. This is in fact true. However, the results in these two sections assume the existence of complete processor to memory connection, i.e., any processor can access any memory module subject only to possible momentary delays due to memory conflicts. As is well known, such a connection network gets very expensive as the system grows and is difficult to expand. The effectiveness of multiprogramming and the distributed scheme depends to a large extent on this complete but expensive connection capability.

Thus, we are interested primarily in the degree of degradation due to monoprogramming, whole module memory allocation, the poorer bandwidth resulting from partitioned and mixed schemes, and various factors which affect this degradation.

In later sections of this chapter we will present similar results

using partial connection networks. We will see the advantages of multi-programming and distributed scheme diminish considerably in these cases.

Before we talk about the first software question, we should first describe some properties of the workload used in the simulation, since the workload will greatly influence system performance. Although we are going to discuss how a change of the workload will affect the result, we feel that a description is needed here in order to give the reader a better understanding of the whole discussion.

3.1.1 The Workload

As we said in Chapter 2, our workload is a sequence of four-tuples. Each four-tuple consists of four pieces of information about a job: the arrival time, the CPU time, the memory requirement, and the number of I/O requests. The original source of this information is the IBM SMF tape. Table 3 displays some statistical data on these parameters, which were obtained by analyzing 1300 real jobs which were run on the University's IBM 360/75 system. Note that we show the mean and the standard deviation of the job interarrival time instead of the job arrival time. This is because the job arrival time is an absolute measurement which does not show the distance between two arrivals unless we lay out all the arrival times. On the other hand, the job interarrival time is a relative measurement which can give us some idea how fast the jobs arrive.

The data in Table 3 are obtained directly from the SMF tape. Sometimes we will scale this data in order to properly load our system. For example, if we want to see the effect of doubling the system load, we can achieve this by reducing the arrival times by one half, thus making it appear as though the jobs are arriving twice as fast. Scaling will be

Data	Mean	σ	Unit
Interarrival Time	6.87	6.96	Sec.
CPU Time	22.68	22.40	Sec.
Job Size	117	80	K bytes
I/O Requests	757	739	No./Job

Table 3. Some Statistical Data of the Workload.

used when we are studying the effect of various workloads.

The real workload, of course, reflects what really happens in a computer system. However, it is very difficult to modify or enlarge. For example, if we want a job stream which is twice as long as what we have now, then we will have to get the second half from the SMF tape and append it to the first half. If we are unlucky, the second half might have completely different characteristics from the first half. For example, if the next day is the due day of a CS101 machine problem, the number of small jobs submitted to the system is suddenly doubled or tripled, which greatly perturbs the characteristics of the workload. This is a very undesirable thing in doing simulation. In addition, it is very difficult to modify some job parameters, e.g., the standard deviations and the distributions. Most of all, it only represents the workload on our 360/75 system, and we would like to see the result of a more general job stream. Therefore, we will use another method we mentioned in Chapter 2, i.e., to produce an artificial workload by using random number generators.

To generate an artificial workload, we have to know the distributions as well as the means and the standard deviations of all four parameters. Of course, we can arbitrarily make up this information. However, in order to maintain some reality, we will obtain them by analyzing the real workload.

Our analysis shows that the distributions of the interarrival time, the CPU time, and the number of I/O requests are approximately exponential functions with the means and standard deviations shown in Table 2. Therefore, we can easily reproduce them by using the following equation:

$$y = -m \log_e 1/(1-x)$$

where m is the mean, x is a uniformly distributed random number in $(0,1)$, and the resulting y is an exponentially distributed random number. The proof can be found in [44].

However, the distribution of the job size is not so simple. Figure 7 shows the density curve of the job size. It contains two bumps, one at around 20K bytes and the other at around 120K bytes. Of course, this depends heavily on the system. The analysis by Chandy, et al [33] also shows the same phenomena. It is very difficult to write down an equation and compute the inverse function, as we did above. We have to do this numerically, i.e., get the cumulative probability function $F(y)$, generate a uniformly distributed random number x in $(0,1)$, and then compute the inverse function

$$y = F^{-1}(x)$$

In fact, this is the basic method of generating a generally distributed random number. However, it is very time-consuming since it involves a searching procedure to determine the interval x as in, and perhaps an interpolation if we want a more accurate value. But, this is all we can do to handle a general distribution. We will use this method for generating the job size.

After knowing these distributions and methods, we can generate four sequences of random numbers to form the artificial workload. This method is very flexible since we can produce a workload with any characteristics we want. Most of the simulations will use the artificial workload.

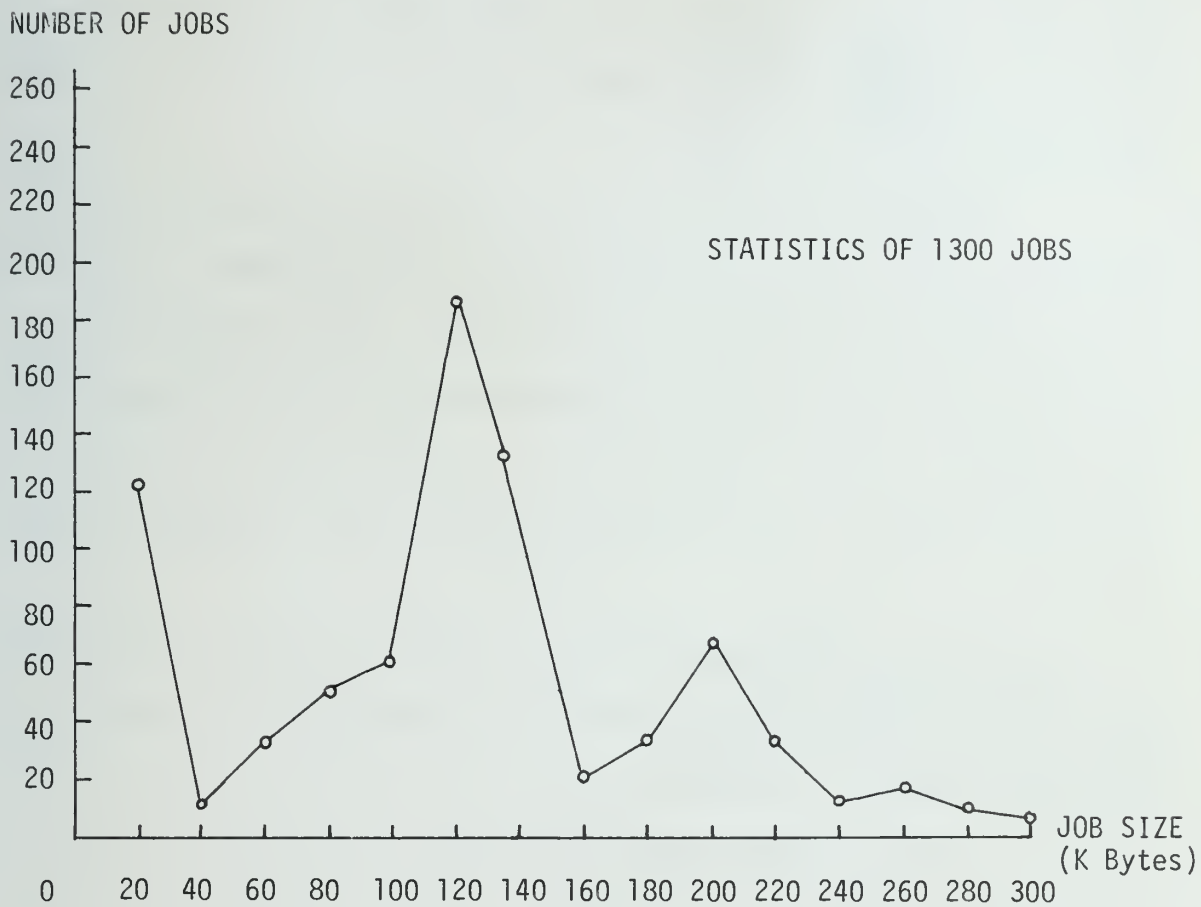


Figure 17. The Density of the Job Size.

Of course, one disadvantage of this method is these parameters are now completely independent of each other. In the real job stream, there might be some correlation among them, for example, a job requiring very large space might use long CPU time and do a large amount of I/O operations. So, there is some difference between the artificial workload and the real workload.

These large jobs will, of course, seriously degrade the average turnaround time. If one of these jobs gets into the memory, it will occupy a large portion of the memory for a long time. This will then block the jobs in the waiting queue from being executed until it finally gets done and releases the memory. Hence, all of the waiting jobs suffer a very long delay which causes a significant increase of the total average turnaround time.

Figure 18 shows some simulation results using both real and artificial workloads, where we use a system with 1024K bytes of main memory. Sixteen memory modules, constant I/O time of 42 ms., memory-processor speed ratio of 4, monoprogramming, shortest-job-first scheduling algorithm, the distributed scheme of memory allocation, a full connection switching network, and 800 jobs. All these parameters were explained in the last chapter and are indicated in the figure.

As we can see from this figure, there is a significant difference between the two curves. After some analysis, we find that this difference is indeed caused by a few very large jobs in the real job stream. Each of these jobs claims a large amount of space and requires a large CPU time, so they contribute a significant amount of queueing delay to the final average turnaround time. However, this does not happen in the artificial

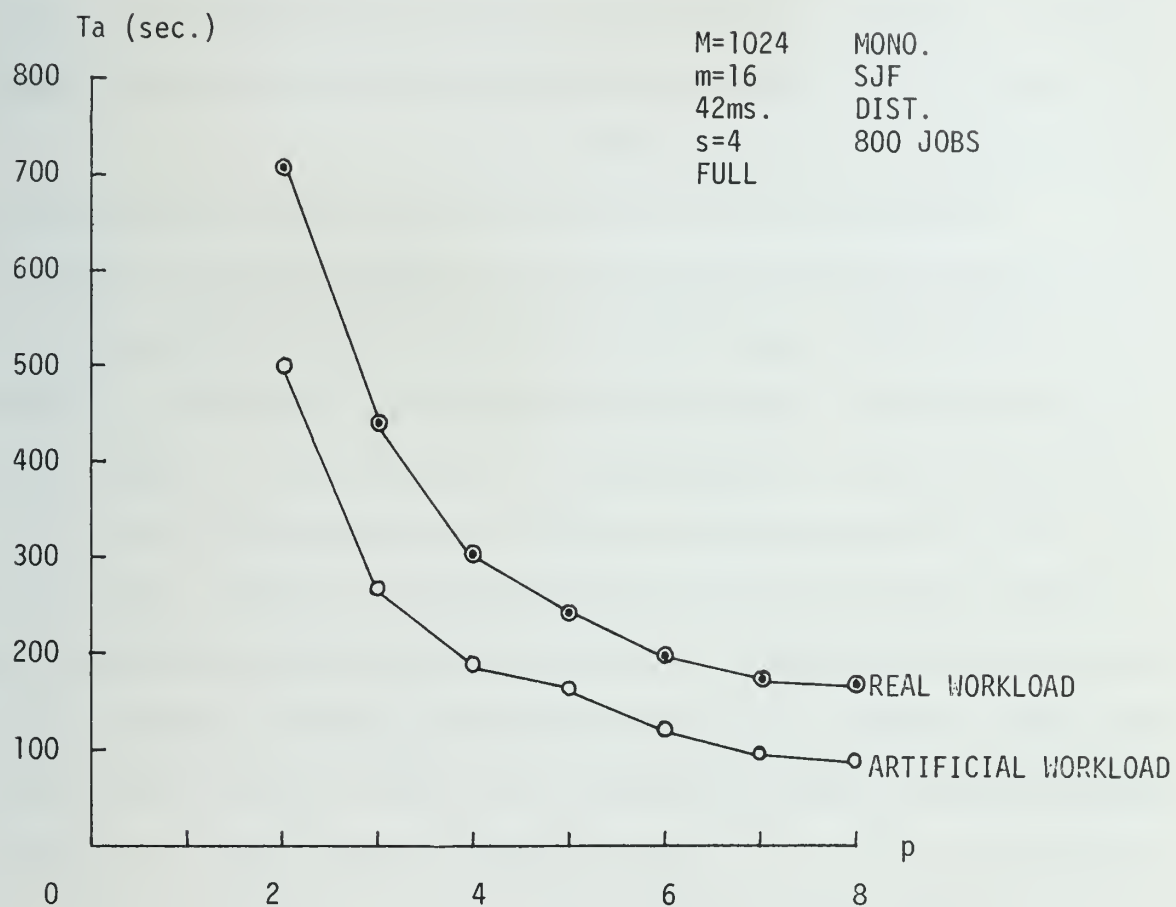


Figure 18. A Comparison of the Turnaround Times of the Real and Artificial Workloads.

workload.

If we delete these big jobs and run the simulation again, we get a result which is just a little bit higher than the result using the artificial workload. Therefore, we believe that the artificial workload is a pretty good approximation of the real workload if we ignore a few big jobs which occasionally occur in the job stream.

However, we are not saying that we will ignore the existence of these big jobs. Actually, this kind of job always exists in a typical university workload. Most of them are so-called number-crunching jobs since they require a large amount of floating-point operations. A number-crunching job needs a processor with fast floating-point arithmetic unit for a fast, efficient execution. Most of the minicomputers and the microprocessors, however, do not provide floating-point hardware. The floating-point operations are done by software or microprogrammed subroutines. So, the number-crunching jobs are not suitable for these small machines. Although a few minicomputers which came out recently do have floating-point hardware, e.g., PDP 11/70, they still cannot provide a fast and efficient execution for this type of job.

The best way to handle these big jobs is to use a big machine like the CDC 7600 or Amdahl 470. These machines all have fast pipe-lined floating-point arithmetic units, which can execute a number-crunching job very quickly and efficiently. This is why these big machines are important in the computation world.

Although a minicomputer or a microprocessor is not appropriate in handling these big jobs, we should not consider this as a fatal disadvantage of building multiprocessor systems by using minicomputer or

microprocessors. We can easily solve, actually we should say "avoid," this problem by the following method. Whenever a big job arrives, we can send it to a big machine elsewhere via a computer network, which can give better service to this job. This is why we use the artificial workload since it approximates the real workload without those big jobs.

In fact, it does not matter which workload we use in our simulation, since we are doing comparison work or finding the effect of a certain parameter. However, the artificial workload seems to be more convenient for us to use, and so we will use it in the following discussion. The real job stream will only be used to provide the necessary information, e.g., mean, variance, and distribution, in generating the artificial job stream.

One thing we would like to point out here is the absolute value of a certain measurement, for example, 150 sec. average turnaround time in general does not have too much meaning alone. Only the relative magnitude or the percentage of difference can indicate the goodness of a certain system over the other one. We will try to use percentages in our presentation.

3.1.2 Monoprogramming versus Multiprogramming

Our first problem is to compare monoprogramming and multiprogramming. As we said in Chapter 1, we are more interested in monoprocessing systems, i.e., each job can only be run on one processor like in the PRIME system. Hence, we are not talking about ILLIAC-IV type machines. Using the definition of Flynn [45], we are dealing with MIMD type of machines, not SIMD type of machines.

Again, we want to remind the reader that all the results in Section 3.1 are using a full connection.

By monoprogramming, as was defined in Chapter 1, we mean each processor is dedicated to a job once it is assigned to this job, and can execute only this job until it is finished. When this job is doing I/O, the processor remains idle. In other words, no processing and I/O overlapping will take place. Due to this rule, we only allow at most as many jobs as the number of processors in the system.

On the other hand, in a multiprogramming system, we will pack as many jobs as possible in the memory and let these jobs share all the processors. Once a processor becomes free, it will try to obtain a job from the processor queue and execute it. No processor will be left idle intentionally if there is a job ready to be executed.

Of course, multiprogramming can result in higher processor utilization and memory utilization, which means higher system throughput. This in general implies we can get shorter average job turnaround time (T_a). Figure 19 shows T_a versus m curves for both monoprogramming and multiprogramming. For $p=4$, the gap between two curves is very big, monoprogramming is about 60% (182 vs 114) higher than multiprogramming. However, when $p=6$, the gap closes to about 13% (102 vs 90).

This is not surprising, since when we increase the number of processors we also increase the maximum number of jobs allowed in the memory in monoprogramming. Apparently, for six processors and a total memory size of 1024K bytes, monoprogramming is already competitive with multiprogramming.

Figure 20 shows how the monoprogramming curve approaches the multiprogramming curve when we increase the number of processors. As we can see, for small p , multiprogramming does show a superiority over

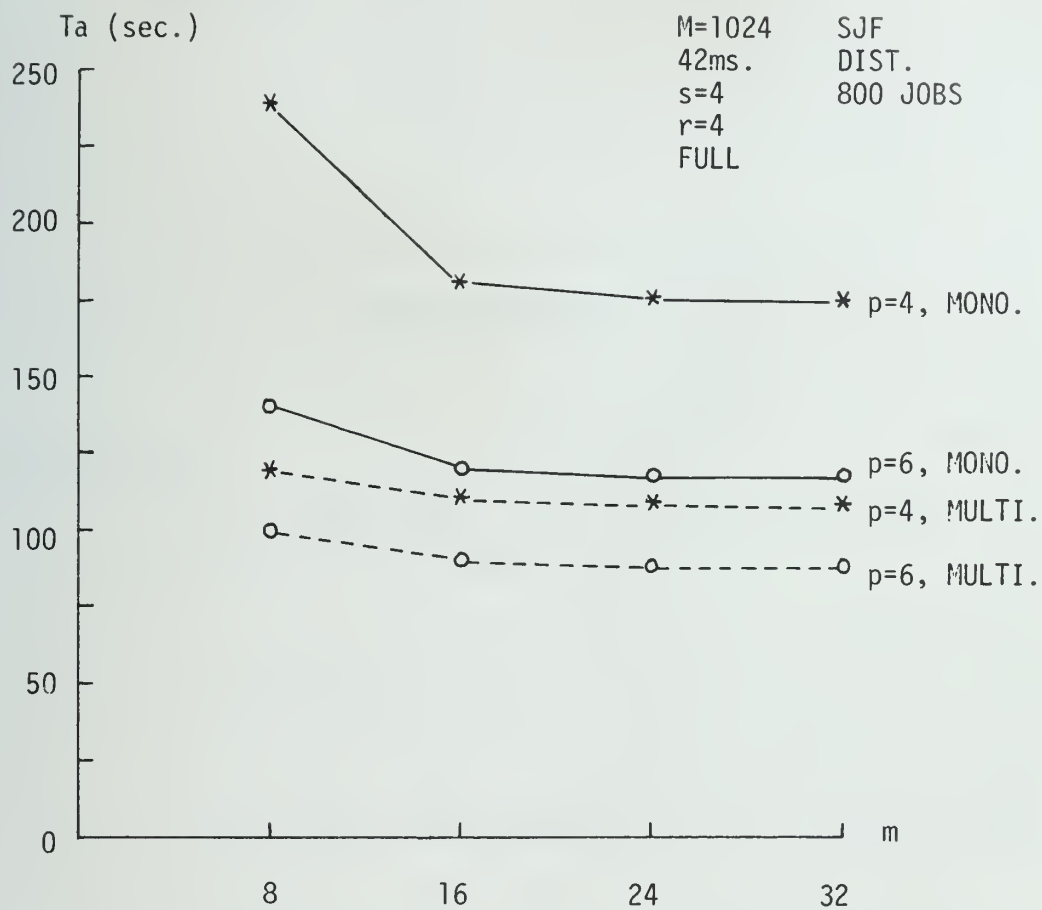


Figure 19. Comparison of Monoprogramming and Multiprogramming.

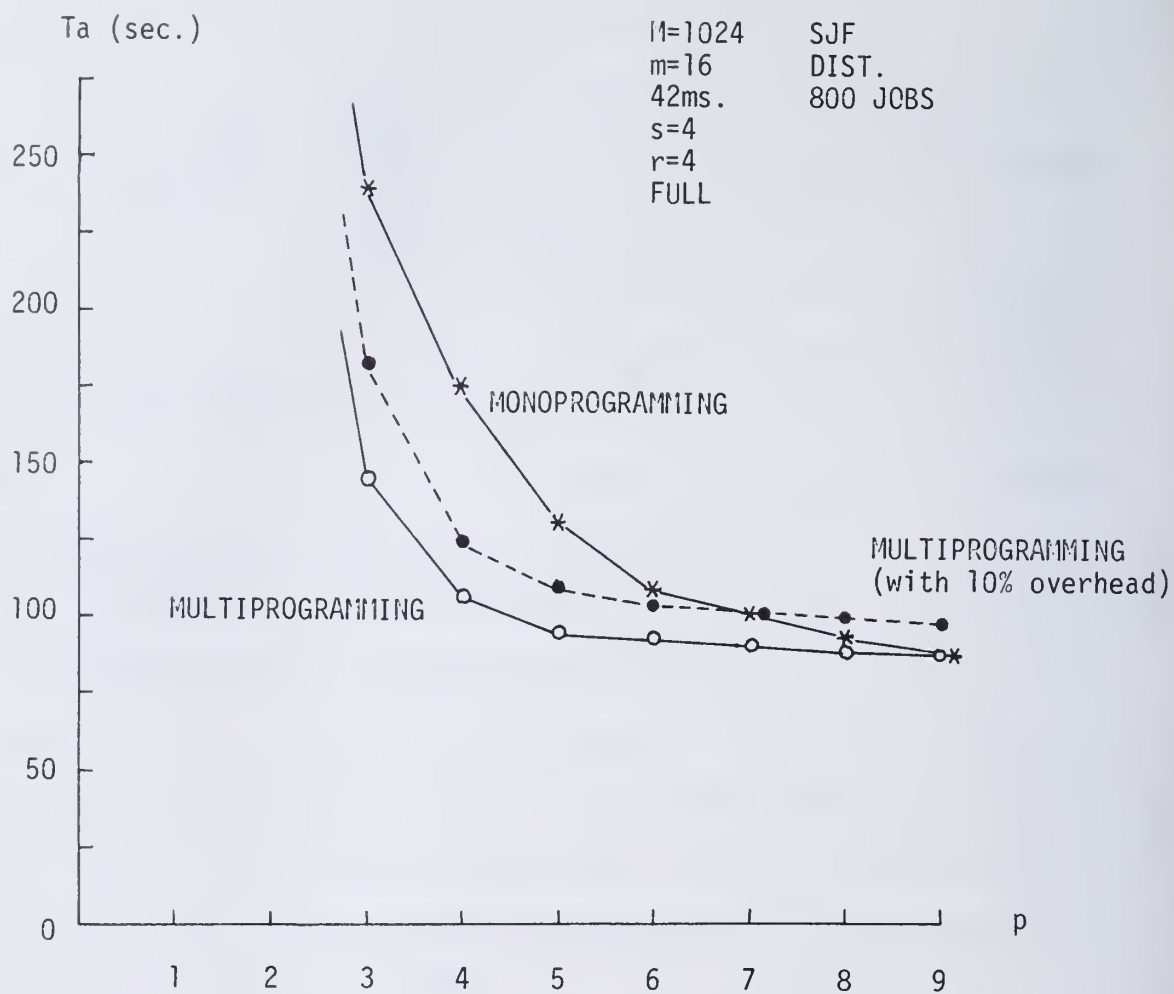


Figure 20. The Effect of Increasing the Number of Processors.

monoprogramming. But with a moderate number of processors, e.g., 6 in this case, two results are more or less the same already. The reason is rather simple, since for the job size distribution we use, the main memory can only contain about six jobs for most of the time. Hence, there is really no reason we should use multiprogramming if we can afford "enough" processors. Of course, "enough" is determined by the total main memory size and the distribution of the job size.

Besides, we have not taken the software overhead of multiprogramming into account. By overhead here, we mean the extra work the multiprogrammed operating system must do, e.g., updating the outgoing user's file, restoring the incoming user's status information, etc. We do not know exactly how high this overhead will be. However, in most computer systems a large portion of CPU time is spent in the operating system. In other words, the processors will have to do more work in a multiprogrammed system than in a monoprogrammed system. If we assume this overhead will increase 10% of the job CPU time, then the T_a curve for multiprogramming will move up to the dotted curve shown in Figure 20. Now, monoprogramming wins for $p > 6$.

The effect of this 10% overhead is different for each p . It causes a larger increment of T_a for smaller p . For larger p , say 8, the increment is about 10%, and for smaller p , say four or less, the increment is more than 20%. Apparently, the overhead is very important when the number of processors is small.

This phenomenon can be explained by Table 4, where for each p value we show the degree of multiprogramming and the queueing delay due to no available processor. The degree of multiprogramming is defined to be the average number of jobs each processor will have to take care of at the

p	Degree of Multiprogramming	Queueing Delay Due to No Processor	Queueing Delay/Average Service Time (%)
2	4.22	85	57.7
3	2.61	41	39.0
4	1.71	20	23.3
5	1.35	13	17.3
6	1.17	10	15.4
7	1.08	9	14.3
8	1.03	8	11.2

Table 4. Degree of Multiprogramming and Queueing
Delay Due to No Available Processor.

same time. In other words, on the average there will be p^* (degree of multiprogramming) jobs in the memory which are sharing p processors. From Table 4, we can see the degree of multiprogramming is high when p is small. This means a lot of jobs will have to compete for a few processors, and the queueing delay caused by waiting for a free processor will be very high. In the second and third columns, we show the queueing delay and its percentage in the total service time. We can see the queueing delay of a two-processor system occupies almost 60% of the total service time and is more than ten times that of an eight-processor system. If we add an overhead to each job, the queueing delay will grow as a function of the degree of multiprogramming since each job will cause a certain amount of delay (the increment of the CPU time) to every job waiting for a processor. This means that the overhead will result in longer delay for smaller p . Therefore, the average turnaround time will increase more for smaller p . We will further explain this in Section 3.1.5.

Before we add the overhead, we can see that multiprogramming wins by a wide margin when p is small. However, the difference will be reduced significantly if we just add a moderate amount of overhead. The increment of the average turnaround time is very sensitive to the overhead, especially for small p . Therefore, the superiority of multiprogramming in that region will disappear rather quickly if the overhead goes up. If we insist on using multiprogramming, the software design will be extremely important. A bad design can easily degrade the performance seriously.

From Figure 20, we can see another interesting point about these curves. If we do not consider the overhead, the multiprogramming curve is only two processors to the left of the monoprogramming curve. If we

use two more processors in the monoprogrammed system, the monoprogramming curve will be shifted to the left by two processors. Thus, two curves will almost overlap. In fact, in this case, the monoprogramming curve will be slightly under the multiprogramming curve. This means that if the difference of the software costs is less than the cost of two processors, then we should use monoprogramming with two more processors. Although we do not have exact figures of these costs so we can draw any conclusion, this apparently is the case in the current trend since the software cost is soaring up rapidly and the hardware cost is going down significantly every year.

If we do take the 10% overhead into account, we can see the difference is only about one processor. Suppose the overhead is even higher, the monoprogramming curve might become completely superior to the multiprogramming curve.

However, we are not completely against multiprogramming. In some cases, multiprogramming is still a better solution for system design. For example, if we have a job mix with all small and I/O-bound jobs, then multiprogramming might give us better results. By I/O-bound, we mean a job which spends most of its life time in doing I/O and relatively small amount of time in processor. In other words, an I/O-bound job will have an I/O time which is, say, several times longer than its CPU time. Most COBOL programs, for example, are I/O-bound under this definition.

The job mix we are using, however, is not I/O-bound. This can be seen from the mean values we show in Table 3. If we assume an I/O operation takes 42 ms, then on the average, a job will spend about 30 seconds in doing I/O, which is of the same order as the average CPU time. Of course, the real average CPU time will not be the same as that shown

in Table 3, since it will be affected by several factors, e.g., the memory allocation scheme (to be discussed in the next section), the degree of interleaving, the memory and processor speeds, etc. However, our simulation results show that it ranges from 30 to 65 seconds. So, the CPU time has been increased by a factor of 1.35 to 2.9. This is caused by several factors. We assume a processor cycle (pc) to be the average amount of time between two successive memory references issued by a processor. In other words, a processor will generate one memory reference in one processor cycle. Since s is defined to be the ratio between the memory speed and the processor speed, a memory cycle (mc) will be $s * pc$. Let us also assume that one CPU second is equal to w processor cycles, or w memory references. So, a program needs $10w$ memory references if its CPU time requirement is 10 seconds. If the average memory bandwidth a processor can get is b , it will take w/b memory cycles to satisfy one CPU second of work. The average memory bandwidth b in general will be less than s due to memory interference. Since $pc = 1/w$ second by definition, w/b memory cycles is equivalent to s/b seconds. Since b is bound by s , s/b will always be greater than 1. That is, it always takes more than one second of time to complete one CPU second of work. This explains why the average processing time ranges from 30 to 65 seconds, instead of being 22.68 seconds shown in Table 3. Therefore, the job mix we are using is not I/O-bound since on the average a job will spend roughly the same or more time in execution than doing I/O.

If we use the job mix with all small and I/O-bound jobs, the results will be quite different from that of Figure 20. Table 5 shows the result by using a new job mix, where we increase the number of I/O requests of each job by 50% and reduce the CPU time and the job size by 25% each.

p	Ta		
	Multi- programming	Multiprogramming (With 10% Overhead)	Mono- programming
4	90	94	671
5	86	90	204
6	83	87	132
7	83	88	107
8	84	88	90

Table 5. The Results by Using New Job Mix.

The average I/O time is now about twice as large as the average CPU time (46 to 24). Therefore, the new job mix is I/O-bound with smaller job sizes. We only show the results for $p=4$ to 8. As we can see, the gap between multiprogramming and monoprogramming is rather large even for p as large as 7. If we add in 10% overhead as we did earlier, multiprogramming (middle column of Table 5) still wins by a slight margin for $p=8$. Apparently, multiprogramming will yield better results for the job mix which contains small and I/O-bound jobs.

Therefore, which strategy we should use depends on the job mix we are dealing with. Our conclusions in this report are all based on the job mix we described in the early part of this chapter, which is a typical workload on a university batch system.

Table 6 shows the memory utilization and processor utilization of both monoprogramming and multiprogramming for the system described in Figure 20. The memory utilization of monoprogramming goes up as p increases, and the memory utilization of multiprogramming essentially remains the same. This is what we would expect. The processor utilizations, on the other hand, both decline as p increases. This is because U_p is the utilization of one processor, in other words it is the "normalized" utilization. If we multiply U_p by p , we can see the results are increasing. This shows the same trend as the T_a curve in Figure 22. Therefore, we can think of $U_p * p$ as a representation of the work being done by the system in a certain time unit.

In fact, the processor utilization is strongly related to the system performance. If $U_p * p$ is higher, the processors can finish more work in one unit of time, and the average turnaround time will be lower.

p	Monoprogramming			Multiprogramming		
	U_m	U_p	$U_p * p$	U_m	U_p	$U_p * p$
4	.44	.50	2.00	.69	.65	2.60
6	.59	.44	2.64	.64	.45	2.70
8	.61	.34	2.72	.62	.32	2.72

Table 6. Hardware Utilizations for Monoprogramming and Multiprogramming.

Therefore, T_a can indirectly tell us what the relative processor utilization should be. In the following discussion, we will not show the utilizations except for a few occasions, since they will act like that in Table 6.

The other measurements that can also indicate the work being done by the system is the total memory bandwidth B_w^T . The total memory bandwidth is the memory bandwidth generated by all the active processors, i.e., processors that are executing jobs. Like $U_p * p$, the higher the total memory bandwidth is, the faster the processors will operate. Figure 21 shows B_w^T versus p curves for the system of Figure 20. As we can see, the total memory bandwidths of the monoprogrammed and multiprogrammed systems both go up as we increase the number of processors. The multiprogrammed system has a higher total memory bandwidth. However, when p is large, say 8, the total memory bandwidths of both systems are very close to each other. Recall Figure 20, the average turnaround times of both systems also become very close to each other as p gets large.

3.1.3 Memory Allocation Schemes

As we described in Figure 5 of Chapter 1, we are interested in three kinds of memory allocation scheme: the partitioned scheme (Figure 5-a), the distributed scheme (Figure 5-b), and the mixed scheme (Figure 5-c). We briefly explained there how they work and their advantages and disadvantages (Table 2). In this section, we are going to investigate their performance and look at some problems related to these schemes.

As we shall see, the memory allocation scheme affects performance in three different ways. First, the space efficiency of an allocation will affect the number of jobs which can be in the memory at any time. The partitioned scheme tends to waste memory since memory can only be allocated

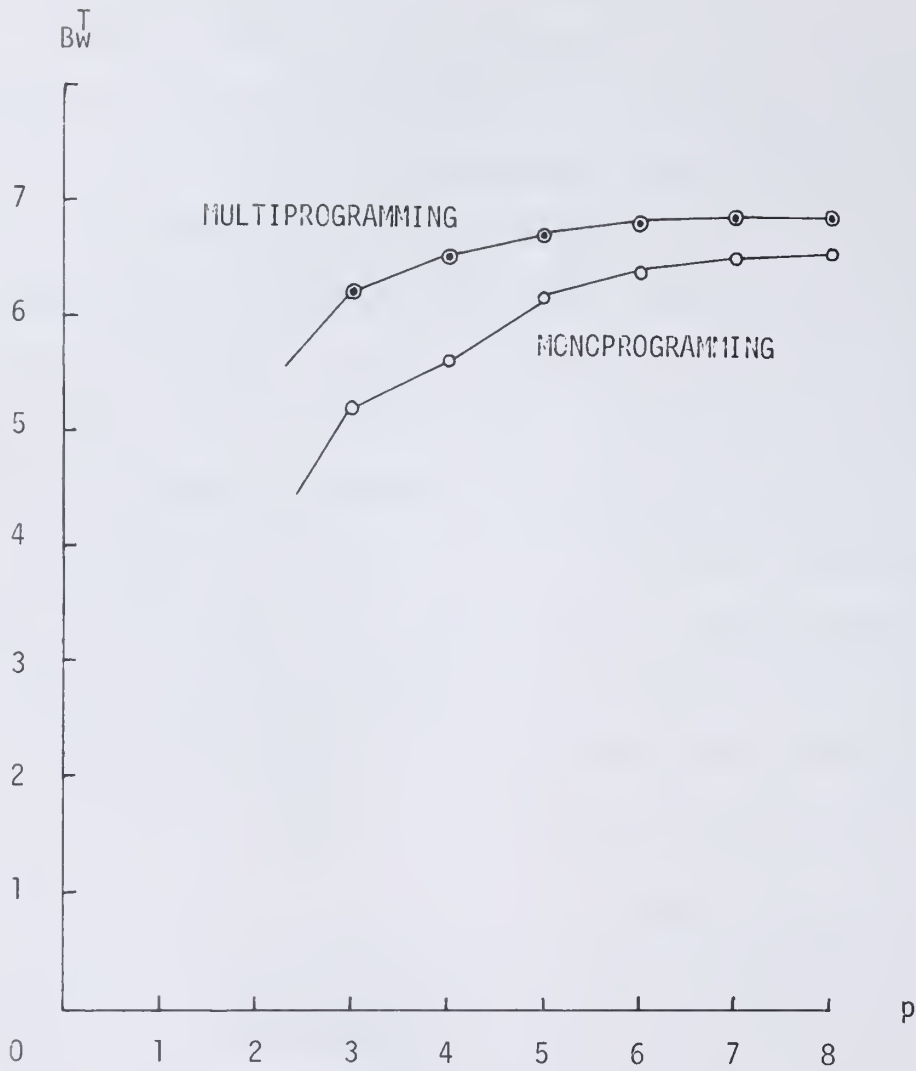


Figure 21. The Total Memory Bandwidth for the System of Figure 20.

in whole modules. The other two allocation schemes do not waste any memory in this way. Hence, the partitioned scheme has less potential to pack jobs in the memory than the other two schemes. Second, the allocation scheme affects the memory bandwidth available to any given job. If, for example, a job requires a small amount of memory (less than one module), then under the partitioned scheme or the mixed scheme only one module will be allocated to this job and the memory bandwidth is limited to 1. On the other hand, distributed allocation causes the job to be spread across all memory modules thus allowing a higher potential bandwidth (although this bandwidth is subject to interference from other jobs in the memory). Suppose, however, that the job requires four memory modules. It may well get worse bandwidth using the distributed scheme than that using the partitioned scheme since the latter case is not subject to interference from the other jobs. Finally, the third effect of allocation on performance has to do with the classical problem of address interleaving which affects the ability of a job to utilize the potential memory bandwidth. This question has been discussed extensively in the literature [39,42,46,47,48]. We see no way of providing definitive answers in this area short of using actual address streams. But, this would lead to results of questionable generality and would be prohibitively expensive. We will, however, establish some bounds on the possible effects of good versus bad address interleaving.

The factors above interact in complex and frequently unpredictable ways. We will attempt to isolate the effects of each factor as much as possible. We begin by analyzing the effect of memory waste on overall performance.

Figure 22 shows the curves of the average turnaround time versus

the number of memory modules for all three schemes, and Figure 23 shows their corresponding total memory bandwidth curves. Notice that the total amount of memory does not change, m is simply the number of modules into which this total is divided. The solid lines represent the monoprogramming curves and the dotted lines represent the multiprogramming curves. Only five curves are shown in both figures; the two curves for the distributed scheme are very close to each other and only one is shown.

As we can see, m has a great influence on both partitioned and mixed schemes, especially from 8 to 16. For $m=8$, the turnaround time of the partitioned scheme is almost ten times as large as that of the distributed scheme. This is very easy to understand. Since with so few memory modules, a large portion of the memory can be easily wasted and not too many jobs can be in the memory at the same time. For example, a job requiring 130K bytes memory will occupy two modules since the module size is 128K bytes, and almost one-eighth of the useful memory has been wasted by this job. Therefore, a job in general has to spend a lot of time in the waiting queue before it finally has enough memory modules to enter the memory.

Table 7 shows the total memory bandwidth (B_w^T), the average job bandwidth (B_w), the memory utilization (U_m), the average number of jobs in memory ($\bar{n}$), and the queueing time of each job ($\bar{q}$) for these three allocation schemes. The average job bandwidth is the average memory bandwidth each job can get while it is being executed. It depends on how we allocate the memory, how we interleave the program, and the speed ratio s of memory and processor. We have described how we calculate the memory bandwidth in the last chapter. Since a processor can only generate up to s memory

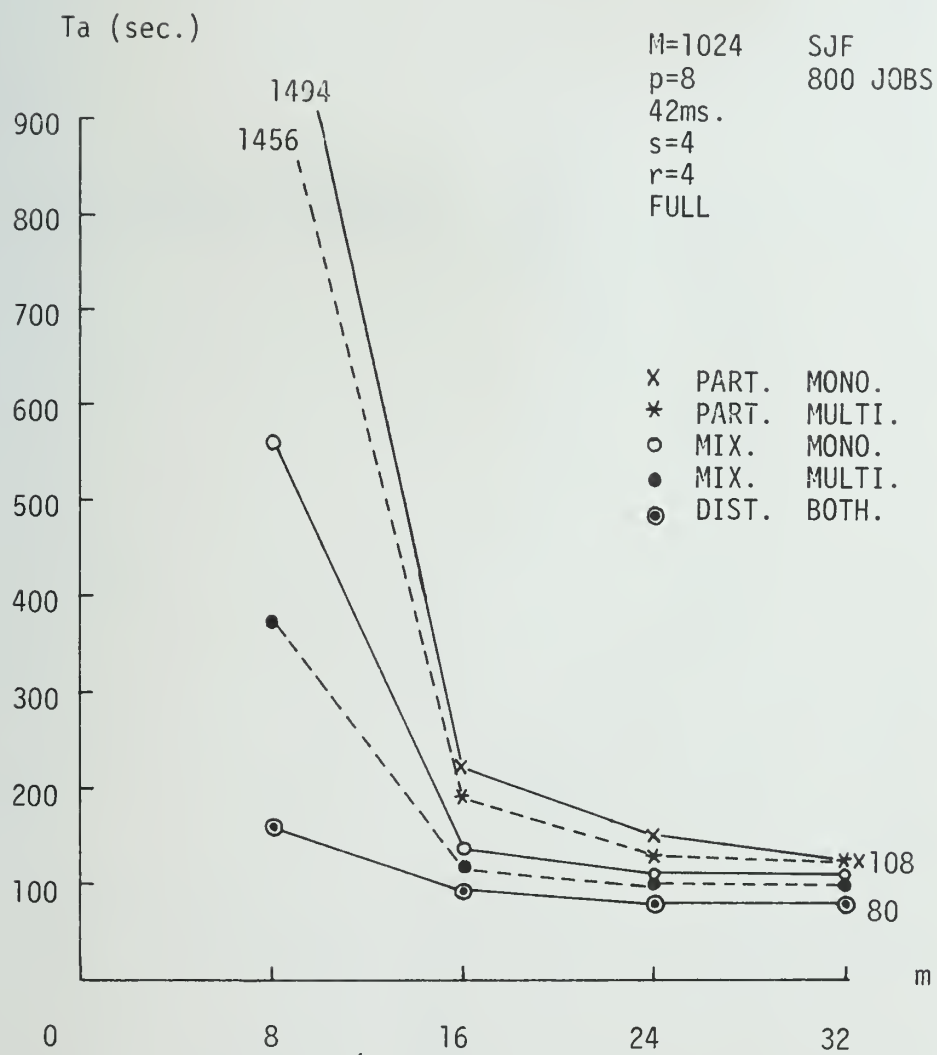


Figure 22. Average Turnaround Times of Three Memory Allocation Schemes.

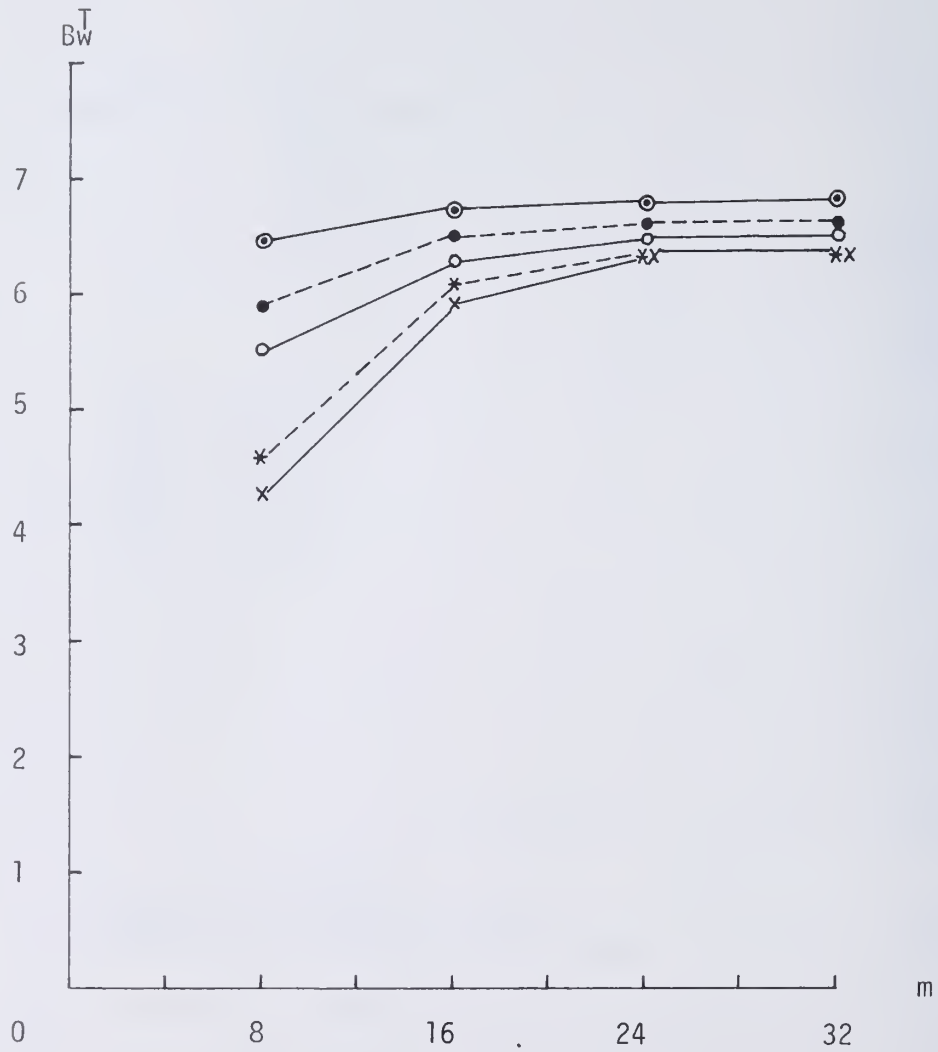


Figure 23. Total Memory Bandwidths of Three Memory Allocation Schemes. (For System in Figure 22)

Scheduling Algorithm		m=8	m=16	m=24	m=32
Partitioned	B_w^T	4.47	6.43	6.51	6.52
	B_w	1.15	1.54	1.75	1.97
	U_m	0.58	0.67	0.62	0.60
	$\bar{n}$	5.5	6.8	6.4	6.2
	$\bar{q}$	1343	112	41	25
Mixed	B_w^T	5.57	6.50	6.54	6.56
	B_w	1.10	1.53	1.76	1.98
	U_m	0.79	0.68	0.63	0.60
	$\bar{n}$	7.6	6.9	6.4	6.1
	$\bar{q}$	445	41	26	19
Distributed	B_w^T	6.50	6.64	6.78	6.82
	B_w	1.46	2.57	2.81	3.18
	U_m	0.80	0.62	0.60	0.57
	$\bar{n}$	7.2	5.5	5.3	5.1
	$\bar{q}$	58	15	12	10

Table 7. The Total Memory Bandwidth (B_w^T), Average Job Bandwidth (B_w), Memory Utilization (U_m), Average Number of Jobs in Memory ($\bar{n}$), and Average Queueing Time ($\bar{q}$) of Three Memory Allocation Schemes. (For the Monoprogrammed System of Figure 22)

requests per memory cycle, B_w is upper bounded by s . From Table 7, we can see the highest job bandwidth we obtain is 3.18, where we use the distributed scheme, 32 memory modules, and a speed ratio of 4.

The job bandwidths for the partitioned scheme and the mixed scheme are very similar. Basically, it is because the ways these two schemes allocate module to a job are quite similar, as we can see in Figure 5. A job will be stored in the same number of modules under both schemes, although there might be some module sharing in the mixed system. Thus, a job is confined in a fixed number of modules and can only access these modules no matter which scheme we use. Most of all, most or all of the job is free from interference by other jobs. There is no memory interference between jobs in the partitioned system, and small interference in the mixed system since only the "overflow" parts will share a module. Therefore, the job bandwidths for these two schemes will be quite close due to these facts. However, the mixed scheme yields better turnaround times, since it can use the memory more efficiently and hence allow more jobs to be processed at the same time. (This tells us that the job bandwidth alone cannot be used to compare the performance of two systems.)

The job bandwidth of the distributed scheme, on the other hand, is much better than the job bandwidths of the other two schemes. The distributed scheme will spread out every job across the whole memory, thus providing each processor the potential of referencing every memory module. Although all the jobs are sharing the memory and the mutual interference is large, a large bandwidth still can be obtained through the large degree of interleaving. Later in this section, we will explain why the distributed system can produce a higher bandwidth than the mixed system by using a numerical example.

The memory utilization in Table 7 is defined to be the percentage of the memory that is actually used by jobs. For the mixed scheme and the distributed scheme, there will be no problem since a job will be allocated exactly the amount of memory it asks for. As for the partitioned scheme, it is a little bit more complicated since the memory is allocated by the module, and in general a job will get more memory than it really needs. So, there are two different memory utilizations we should distinguish. One is the utilization we defined above, which is to calculate the percentage of the memory really used by the jobs. The other one is the percentage of the memory that is occupied by the jobs. Of course, the latter is larger than the former since some memory will be occupied but not be used. In other words, some memory is wasted under the partitioned scheme.

In order to distinguish these two types of memory utilization, we will call the first one "word memory utilization" and the second one "module memory utilization." Of course, both types of memory utilization will be the same in a mixed system or a distributed system, i.e., the word memory utilization. The utilizations we show in Table 7 are all word memory utilizations. Most of the time we will just call this memory utilization for short.

One interesting thing is that the difference of these two memory utilizations is the percentage of the memory which has been wasted, i.e., occupied but unused. This is very easy to understand. We will show some results of the memory waste of the partitioned scheme later.

As we can see from Table 7, for $m=8$ and under monoprogramming, the partitioned scheme has a total memory bandwidth of 4.47, a job bandwidth

of 1.15, a (word) memory utilization of only 58%, and averages 5.5 jobs in the memory, which results in an average queueing time of 1343 seconds! Under the same condition, the distributed system has a total bandwidth of 6.50, a job bandwidth of 1.46, a memory utilization of 80%, averages 7.2 jobs, and has an average queueing time of only 58 seconds.

Of course, one way to improve the performance of the partitioned system is to increase the number of memory modules and to decrease the size of each module. This can reduce the amount of wasted memory, since on the average each job will waste one-half of a module (see the proof in Chapter 2). Thus, the probability that a job gets blocked due to insufficient memory will be reduced. In Table 8, we show the word utilization, the module utilization, and the memory waste of the partitioned system in Figure 22. As we can see, when we double the number of modules from 8 to 16, the word memory utilization of the partitioned system increases to 67%, and the module memory utilization drops a little down to 92%. Meanwhile, the memory waste has been reduced from 37% to 25%. This is why the average queueing time reduces sharply to 112 seconds (a gain of 12)! If we further increase the number of modules to 24, the memory waste will decrease to 14%, and the average queueing time down to 41 seconds. Apparently, the partitioned system is very sensitive to the number of modules. The main reason is, of course, the memory waste will reduce the ability of accepting jobs. Therefore, it is very important to provide enough memory modules in the partitioned system.

Actually, the memory utilization and the average number of jobs in memory do not grow as the system performance improves. On the contrary, all the memory utilizations and the average numbers of jobs decrease when

	m=8	m=16	m=24	m=32
Word Memory Utilization	0.58	0.67	0.62	0.60
Module Memory Utilization	0.95	0.92	0.76	0.71
Memory Waste	0.37	0.25	0.14	0.11

Table 8. Memory Waste for the Partitioned Scheme.

we increase m , except for the case we just mentioned. And surprisingly, the distributed scheme has the smallest values of U_m and $\bar{n}$ among these schemes, yet on the other hand, it has the best turnaround time. This means that the higher utilization (as we define it) does not necessarily imply a better throughput.

In fact, U_m and $\bar{n}$ should decrease as the system throughput increases, since if the arrival rate is fixed, the faster the system operates, the faster the jobs leave, and the emptier the system will be.* Especially in a distributed system, the fewer jobs in the memory, the less memory contention each job will suffer, and the higher bandwidth each processor will get to execute a job. The system throughput (the memory bandwidth) goes up when we increase the number of memory modules. This explains why U_m and $\bar{n}$ decrease as m increases.

The distributed scheme, however, does not have this memory utilization advantage over the mixed scheme since both the distributed and mixed schemes can fully utilize the memory. This was shown in Figure 5. In the first column of Table 7, the mixed scheme indeed shows memory utilization (79%) and average number of jobs in memory (7.5) comparable to that of the distributed scheme. Despite this, the distributed scheme still yields a better turnaround time, for every m value. Apparently, the distributed scheme can produce higher bandwidth than the mixed scheme can, if they are given the same set of jobs in the memory. The difference must come from the degree of interleaving a scheme provides to each job, since this is the only difference between these two schemes.

*This can be explained by using Little's Theorem, i.e., $\bar{n} = \lambda \bar{x}$, where λ is the job arrival rate and $\bar{x}$ is the average turnaround time.

Let us look at an example which can explain why the distributed scheme generates higher memory bandwidth than the mixed scheme does. Assuming we have a memory system of 8 modules and four jobs of sizes $1\frac{2}{3}$ modules, $2\frac{1}{2}$ modules, $1\frac{1}{3}$ modules, and 2 modules respectively. These jobs are stored in the memory as shown in Figure 24. The numbers shown in Figure 24-a are the fractions of these jobs in each individual module. We assume they are the reference probabilities we need in the general bandwidth equation, i.e., Equation (6) in the last chapter. Of course, for the distributed system shown in Figure 24-b, all the reference probabilities are $1/8$. Let us also assume that the references generated by a processor are all independent, as we do in our simulation. In order to mimic the real operation, we will in addition assume job a is doing I/O and hence does not contribute to the total bandwidth.

Now, for the distributed system, the bandwidth can be calculated as follows.

$$\begin{aligned}
 B_w^d &= \sum_{j=1}^m \left[1 - \prod_{i=1}^p q_{ij}^{(s)} \right] = m[1 - (1 - 1/m)^{p \cdot s}] \\
 &= 8[1 - (1 - 1/8)^{3 \cdot 4}] \\
 &= 8[1 - (7/8)^{12}] \\
 &= 6.389
 \end{aligned}$$

where $s=4$ is the memory-processor speed ratio which we assumed in Figure 22. As we can see, s contributes a lot in the above calculation since it increases the power of the second term in the parenthesis. As for the mixed system, the bandwidth can be calculated by first finding out $q_{ij}^{(s)}$'s.

Here are all the numerical values.

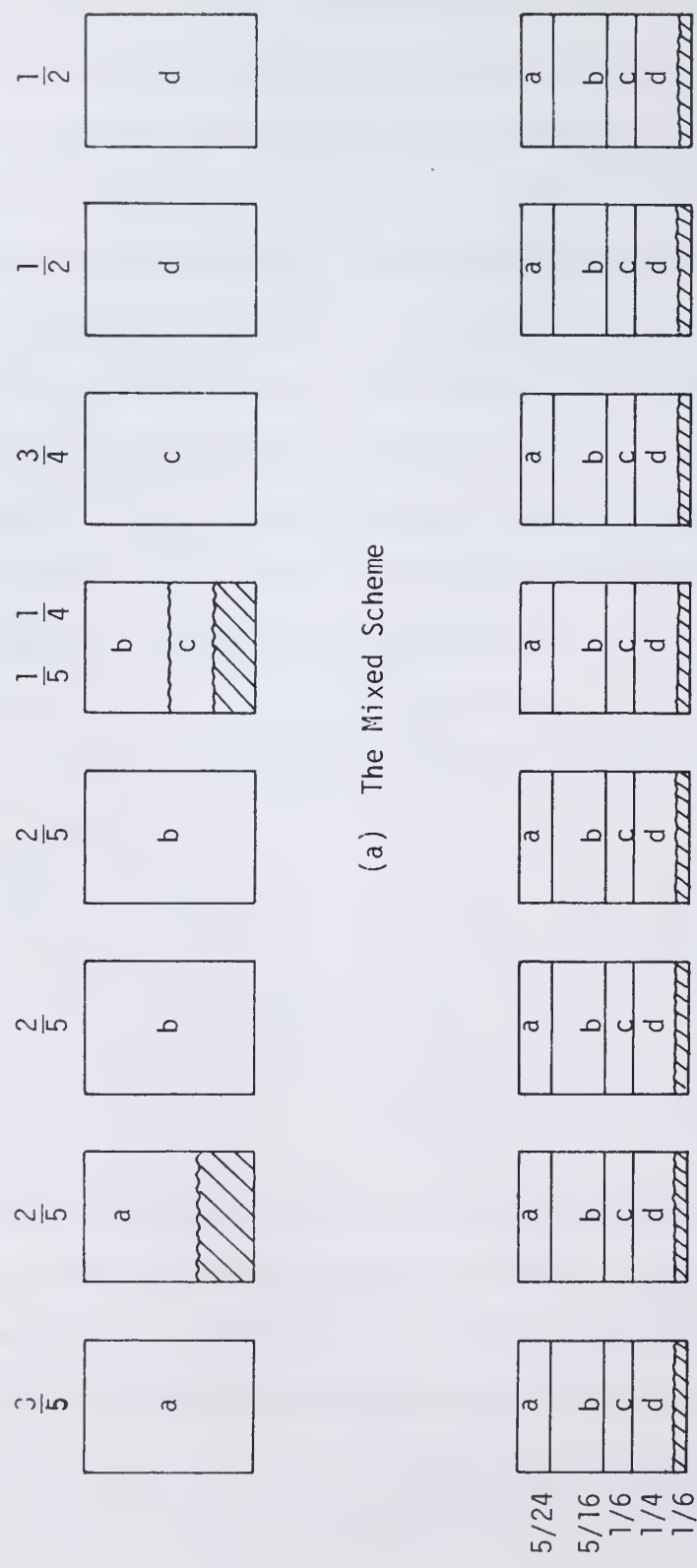


Figure 24. Memory Allocation of the Mixed and Distributed Schemes.

$$q_{11}^{(4)} = 1$$

$$q_{12}^{(4)} = 1$$

$$q_{23}^{(4)} = (1 - 2/5)^4 = 0.1296$$

$$q_{24}^{(4)} = (1 - 2/5)^4 = 0.1296$$

$$q_{25}^{(4)} = (1 - 1/5)^4 = 0.4096$$

$$q_{35}^{(4)} = (1 - 1/4)^4 = 0.3164$$

$$q_{36}^{(4)} = (1 - 3/4)^4 = 0.0039$$

$$q_{47}^{(4)} = (1 - 1/2)^4 = 0.0625$$

$$q_{48}^{(4)} = (1 - 1/2)^4 = 0.0625$$

Using the same equation, we can get the following result.

$$\begin{aligned} B_w^m &= 0 + 0 + 0.8704 + 0.8704 + 0.8704 + 0.9961 + 0.9375 + 0.9375 \\ &= 5.482 \end{aligned}$$

Comparing these two results, we can see the distributed scheme produces 16.5% more bandwidth than the mixed scheme.

In fact, if all four jobs are active, i.e., all four processors are accessing the memory, the mixed scheme can produce a total bandwidth of 7.321, and the distributed scheme only produces 7.055. However, the probability that all jobs are active is rather small, especially if a job spends a significant amount of time in doing I/O like the workload we are using. Suppose more than one job is doing I/O, the distributed system will open an even larger margin over the mixed system, since more modules will idle in the mixed system. Therefore, the distributed system wins

by a large margin most of the time. This explains why the turnaround time of the distributed system is the lowest in these schemes.

In summary, the performance difference between the partitioned system and the mixed system is caused by the job packing, and that between the mixed system and the distributed system is caused by the job bandwidth. This has been carefully explained above and can be seen in Table 7.

Recall Figure 22, we can see all the curves are pulling together as m gets larger. For $m=32$, the monoprogramming curve of the partitioned system is only 35% ($28/80$) higher than the multiprogramming curve of the distributed system. This shows that the partitioned system is comparable to the distributed system if we can afford a large number of modules.

Actually, one very important factor that makes the distributed scheme better than any other scheme is the memory-processor speed ratio s . The second term in the parenthesis of the equation for B_w^d will diminish very fast as s gets larger, which makes the bandwidth approach m (the perfect bandwidth) very quickly. On the other hand, the distributed system will lose its superiority as s gets smaller. Table 9 shows the turnaround time versus m value for $s=2$. This table is the same representation as Figure 22, except we are emphasizing the numerical values this time. As we can see, the monoprogramming result of the partitioned system pulls within 14% when $m=24$.

In the limiting case when $s=1$, the partitioned system will have the best bandwidth if we assume the same number of jobs in the memory. This is because the partitioned system does not have any memory interference between jobs, but the other two have. The only reason the distributed or mixed system can win is that they can fully utilize the memory, and hence

Monoprogramming, Multiprogramming				
System	m=8	m=16	m=24	m=32
Partitioned	186, 191	91, 92	83, 83	80, 81
Mixed	99, 95	83, 82	80, 79	78, 78
Distributed	81, 82	73, 74	73, 73	71, 72

Table 9. T_a Versus m for Three Memory Allocation Schemes.

($s=2$)

the probability that a job cannot enter due to insufficient memory is the smallest. Our simulation result shows that the turnaround time of the partitioned system is higher than the distributed system by only 15% (78/68) when $m=16$, and by only 7% (72/67) when $m=24$. Therefore, the partitioned system does perform very well when s is small.

Currently, the memory technology can provide us a semiconductor with a cycle time of less than 100 ns. If we use a microprocessor with a similar cycle time, then a s value of 1 is realizable. Hence, the use of the partitioned scheme is indeed very favorable since it has so many advantages as we described in Chapter 1, and yet it performs equally as well as any other scheme.

We compared the performances of monoprogramming and multiprogramming in the last section, and we claimed that monoprogramming is very comparable with multiprogramming when we have enough processors. Both Figure 22 and Table 9 again give strong support for this fact. As a matter of fact, some results in Table 9 even show that monoprogramming is slightly better than multiprogramming!

When we decrease the memory-processor ratio s , it could mean we use either slower processor or faster memory. In our simulation, we hold the processor speed constant. So, changing s from 4 to 2 means reducing the memory cycle time by half. From Figure 22 and Table 9, we can see s has a significant effect on the partitioned scheme, particularly for small m . When we reduce s from 4 to 2, the turnaround time improves a lot ranging from 800% to 35%. Of course, we have to pay the price of faster memory. We will come back on this subject later in this chapter.

Although there are several advantages of using the partitioned

scheme (for example, it is very reliable and easy to expand), there are some implementation problems, e.g., address mapping. Let us give a simple example to explain this problem.

Suppose a job requires three modules of memory. How do we store this job in the memory when it gets these modules? We can either three-way interleave this job or store it in a sequential manner, i.e., store the first $1/3$ of the program in the first module, the second $1/3$ in the second module, and the rest in the last module. This second method does not create any particular addressing problem, since the instructions and data are stored sequentially inside a module and the ordinary address generation mechanism can be used to produce physical addresses. So, as long as we know which three modules contain this job, we will have no problem fetching or storing in these modules. Of course, the module size in general will be a power of 2, which makes the address mapping extremely easy. However, this scheme may not allow us to take the advantage of the independent memory modules, since if we assume a serial address stream, only one word will be accessed at a time. This implies we can only get a minimum bandwidth of 1! For $s > 1$, this scheme apparently will waste the processing power due to insufficient memory bandwidth.

In order to get a higher bandwidth, we should use address interleaving, which we assume in our simulator. If we use this scheme, we need a more complicated mapping mechanism to generate the physical addresses, since now the consecutive instructions or data are stored in different modules. The difficulties are: the degree of interleaving is variable depending on the job size, and the modules allocated to the same job might not be adjacent to each other. Hence, we cannot get the next physical

address by simply incrementing the current program counter by one, as we can do in the above non-interleaving scheme or in the distributed system. Of course, indirect addressing and the branching instructions are even more difficult to solve. Therefore, we do need some extra hardware and an address generating algorithm in the instruction unit if we want to use the interleaving scheme in a partitioned system.

We are interested in the hardware design of this problem. In the next chapter, we will discuss a few feasible methods which involve the use of quotient-remainder operation.

From Figure 22 and Table 9, we can see the mixed scheme only out-performs the partitioned scheme by a small margin. However, it is less reliable since the failure of a shared module might affect several jobs. Moreover, it needs a more complicated operating system which is one thing we are against. So, we feel that the partitioned scheme is a very good choice.

In the example of Figure 24, we assume the probability that a processor accesses a certain module to be the fraction of the program that is stored in that module. This assumption obviously is only valid for the random access system, that is, a processor which generates references to the memory modules in a random fashion. In other words, there is no relationship between any two successive references generated by a processor and the second reference has the same chance to refer to any module occupied by this program independent of the first one.

Of course, this assumption about random addressing is not necessarily valid. It is well known that address streams tend to be somewhat

serial. A serial address stream will produce better performance than a random address stream if the addresses are interleaved across several modules, and worst performance if the addresses run vertically in each module. Unfortunately, it is difficult to adequately quantify this seriality or to determine a typical value of seriality. This has forced us to use the random addressing assumption in our simulation.

Now, let us find out how reasonable our assumption will be. Let us look at some performance bounds to see how good and how bad our system will perform if we assume the perfect and the worst memory bandwidth cases. In the perfect memory bandwidth case, we will assume there is no memory conflict between processors, so each processor will be getting a maximum possible bandwidth of s . However, in a distributed system, if the number of active processors (n_p) times s is greater than the total number of memory modules (m), we will assume each processor is only getting a bandwidth of m/n_p ($< s$). In a mixed or partitioned system if s is greater than the number of modules assigned to a job m_j , then the processor will only get a bandwidth of m_j . Of course, we will then get the best possible performance among all the systems that have the same set of system parameters, i.e., a lower bound of the turnaround. This case can only happen if we horizontally interleave every program across the memory modules, and assume a perfect condition of no memory conflict.

On the other hand, for the worst memory bandwidth, we will assume each active processor is only getting a bandwidth of 1, which corresponds to the situation when we vertically store each program inside a memory module (no interleaving at all) and unluckily no two references will ever go to two modules. Of course, this case might not happen but it does give us the

worst performance which can serve as an upper bound on the turnaround time.

Figure 25 repeats the curves of Figure 22 together with four curves which represent the performance of the perfect and worst memory bandwidth cases. We use monoprogrammed systems to derive the two upper bound curves since they contain fewer jobs. On the other hand, we use multiprogrammed systems to obtain the two lower bound curves since they can contain more jobs. As we can see, the monoprogrammed, partitioned system yields the worst result which we call the largest upper bound, and the multiprogrammed, distributed system yields the best result which we call the smallest lower bound. Any performance curve will be bounded between these two curves no matter what the memory allocation scheme we use and how we assume the reference probabilities, as long as we are using eight processors, the shortest-job-first algorithm, an average I/O speed of 42 ms, 1024K bytes main memory, a full connection, and a speed ratio of 4 (cf. Figure 22).

One interesting thing is that when $m \geq 16$, all the curves are clustered above the lower bound curve. Obviously, the random distribution assumption already gives us pretty good results. Further complication of the memory bandwidth calculation apparently can only give a very minor effect on these performance curves. This explains why we are using the random distribution assumption in all our simulations. Furthermore, as we can see, all the curves are far below the upper bound curves. This tells us that the (horizontal) interleaving scheme can be a very important factor on the system performance.

Now, let us briefly summarize the results we have in this section. Table 10 shows an overall comparison of these three memory allocation schemes.

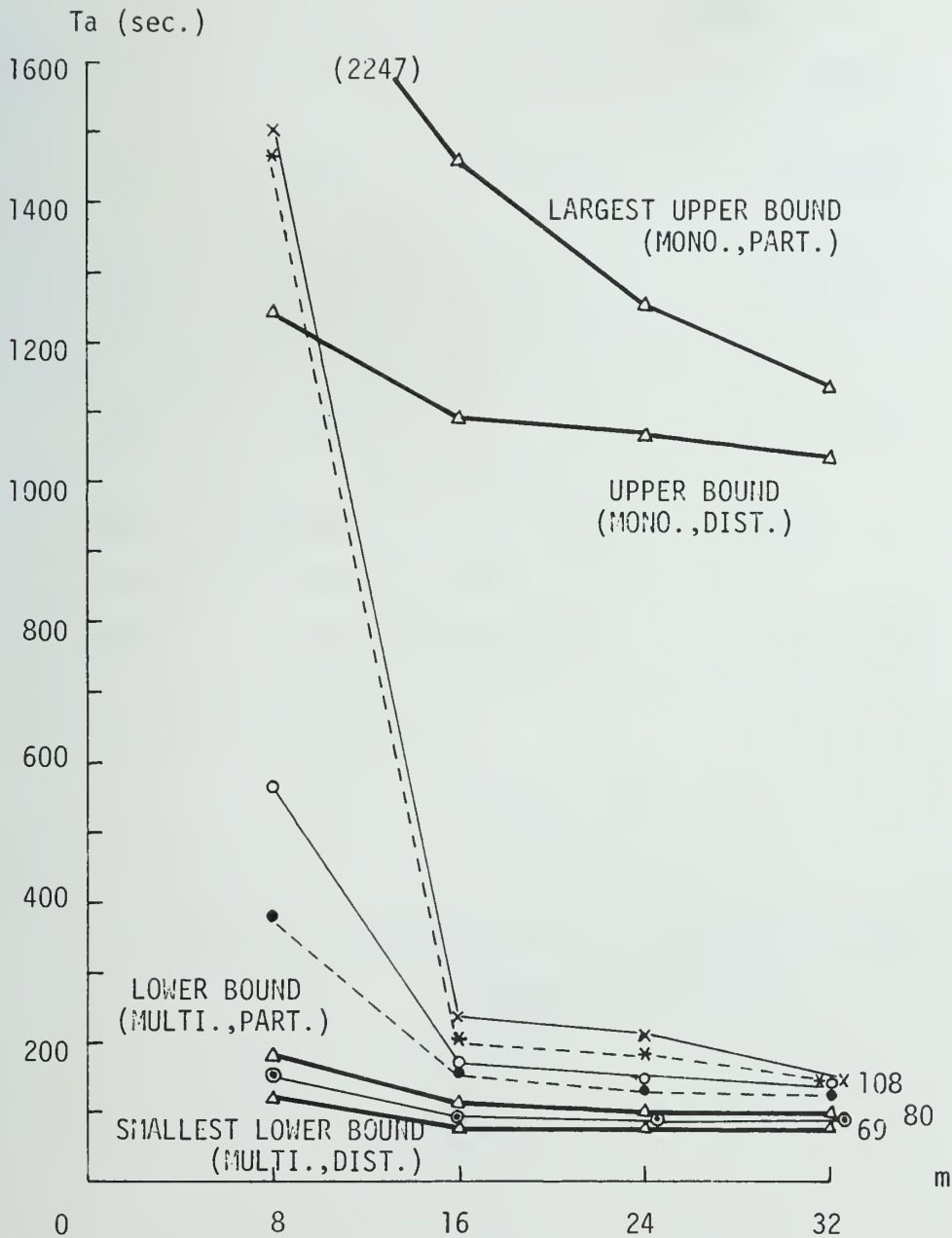


Figure 25. The Performance Curves of Figure 22 and Some Performance Bound Curves.

Parameter \ Scheme	Partitioned	Mixed	Distributed
Total Memory Bandwidth	Moderate	Moderate	High
Job Memory Bandwidth	Moderate	Moderate	High
(Word) Memory Utilization	Bad	Good	Good
Reliability	Good	Moderate	Bad
Turnaround Time	Bad for Small m Good for Large m	Good	Best
Memory Waste	Yes	No	No

Table 10. Summary of the Performance of Three Memory Allocation Schemes.

The distributed scheme leads in all items except the reliability. On the other hand, the partitioned scheme trails in all items, except it is the most reliable one. However, we have shown that when we increase the number of memory modules in the system or improve the memory speed, the performance of the partitioned scheme improves very quickly. For a moderately large number of modules, say 24, the partitioned system already has a performance which is very comparable to the distributed system. Besides, the partitioned scheme provides a very high reliability. In a system where the reliability is extremely important, the partitioned scheme should be considered first. If we need a higher performance and can sacrifice a little bit of reliability, then the mixed scheme might be a better choice. Of course, if we are primarily interested in performance and have very reliable hardware, i.e., the mean time between faults (MTBF) is long, then the distributed scheme will be the best candidate.

3.1.4 Job Scheduling Algorithm

One of the major factors that affect the turnaround time of a job is the scheduling algorithm. The scheduling algorithm is used to determine the order that the jobs in the waiting queue will enter the system. This is based on some attribute of these jobs, not necessarily the order they arrive. Therefore, some new jobs might enter the memory and get executed before a job which arrived previously. This, of course, increases the time this old job has to spend in the waiting queue. But on the other hand, those jobs which get into the memory earlier will suffer shorter queueing delays.

In fact, the purpose of using a scheduling algorithm is to somehow scramble the execution order of a certain set of jobs such that some

kind of performance will be improved. Most of the time, the average turn-around time or the average queueing time will be what people are trying to improve.

The queueing time of scheduling algorithms has been extensively studied by queueing theorists. In [26], Kleinrock has a very complete discussion on this subject. Most of the analytic results are expressed in terms of the average conditional queueing time, i.e., the queueing time of a job which needs a certain amount of processing time. For example, Figure 26 shows the average conditional queueing time curves of three commonly used scheduling algorithms in time-sharing systems, namely FCFS (first-come-first-serve), RR (round-robin), and FB (foreground-background), assuming the system is an M/M/1 queue. We do not show the scales since they depend on the arrival rate, the mean service time and the service-time distribution. Only the shape of a curve is shown which gives the effect of a scheduling algorithm on the jobs with different processing time requirements.

As we can see, the average conditional queueing time for FCFS is the same (constant) for any job, whether it requires long processing time or short processing time. This type of scheduling algorithm is called non-discriminating. FCFS can give the shortest queueing time to long jobs. However, it gives the longest queueing time to short jobs. The average conditional queueing time for RR, on the other hand, grows linearly as the processing time increases. The longer the job is, the larger the queueing time will be. This kind of scheduling algorithm is called linear-discriminating. It discriminates against long jobs. Similarly, FB is called the most-discriminating since it gives the longest queueing time to long jobs among

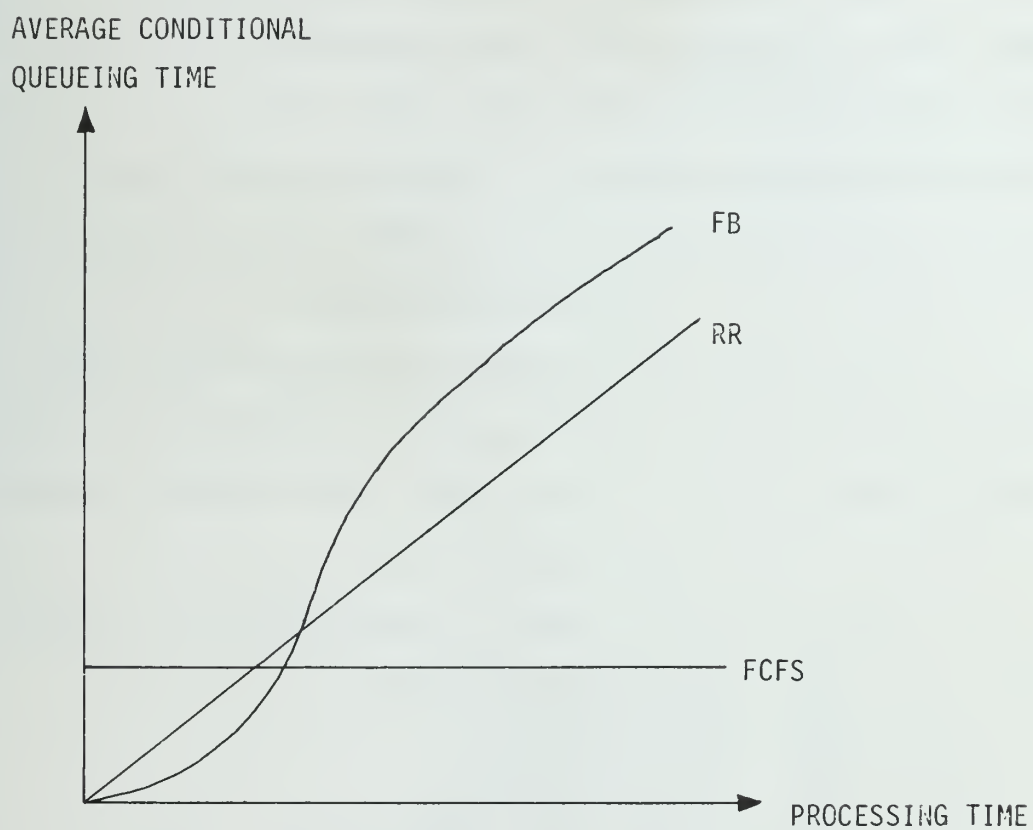


Figure 26. Average Conditional Queueing Time for M/M/1 System.

all known scheduling algorithms. But, FB yields the shortest queueing time to short jobs. RR is a very popular scheme in a lot of time-sharing systems. FB is used in the famous MULTICS system.

There is no absolute standard of which algorithm is the best among these three. All depend on what kind of measurement we are most interested in and the job mix we are dealing with. For example, if we are interested in the overall average queueing time and most of the jobs are short jobs, then apparently we should go for FB. However, since we are dealing with batch systems, we will not use these algorithms.

In our study, we will use the average turnaround time (T_a) of all jobs as our measurement of goodness. This is the same as using the overall average queueing time ($\bar{q}$), since a longer $\bar{q}$ always implies a longer T_a . We will show both of the measures later.

The following eight scheduling algorithms will be studied here.

1. FCFS: first-come-first-serve
2. SJF: shortest-job-first
3. LJF: longest-job-first
4. SMF: smallest-job-first
5. LMF: largest-memory-first
6. SMNF: smallest-magic-number-first
7. SPTF: shortest-processing-time-first
8. BMFF: best-memory-fit-first

All these names are self-explanatory. SMNF is a scheme used in our IBM 360/75 system. Each job is assigned a magic number which is calculated by using the following formula:

$$MN = 3 * (\text{processing time}) + 0.01(\text{job size})^2 + 0.05 * (\text{Number of I/O requests})$$

Then, the job with the smallest magic number will get executed first. This scheme not only penalizes long jobs but also penalizes large jobs (jobs

requiring large memory spaces), since the above formula takes both processing time and job size into account. BMFF will choose the job which can fit into the memory best. If the available memory space is very large, then BMFF will act just like LMF since the largest job in the waiting queue will be chosen. However, if the remaining space cannot hold the largest job, some smaller job which fits best will be chosen instead. SPTF will choose the job with the shortest processing time. It is slightly different from SJF, since SJF also takes I/O time into account. In other words, SJF will choose the job which has the smallest CPU plus I/O time.

When a job arrives, it will be placed somewhere in the waiting queue according to the scheduling algorithm. For example, SMNF will line up the jobs such that their magic numbers are in increasing order. Then, the queue will be considered from the beginning every time.

Table 11-a shows T_a versus p values for all these algorithms, and Table 11-b displays the numerical values of the average queueing time ($\bar{q}$). As we can see, SJF gives the smallest turnaround time among all these scheduling algorithms. This is what we would expect since it has been proven analytically [26]. Therefore, we use SJF in all other discussions.

In fact, for $p > 6$ all these algorithms perform more or less the same. For example, FCFS is less than 10% from SJF. This means when we have enough hardware the scheduling algorithm really does not make too much difference on the performance. It is very easy to understand, since when the throughput is high the system will be lightly loaded, and no matter how we schedule the job each job will only suffer very little delay. Only when the system is heavily loaded will the scheduling algorithm be important. Maybe the following adaptive method can be used. If the system is lightly

M=1024
m=24
42ms
s=2
r=4
FULL

PART.
MONO.
800 JOBS

Scheduling Algorithm \ p	3	4	5	6	7	8
LJF	4663	1707	235	124	95	87
SMF	3415	1354	198	114	94	86
FCFS	3218	1140	179	109	92	84
BMFF	3121	1151	171	106	88	80
LMF	2691	1006	174	120	97	89
SMNF	2604	1029	166	101	87	79
SPTF	2592	1031	160	103	89	80
SJF	1960	776	152	100	88	79

(a) T_a Versus p

Scheduling Algorithm \ p	3	4	5	6	7	8
LJF	4600	1643	169	57	27	19
SMF	3352	1291	133	47	26	18
FCFS	3065	1077	114	42	24	17
BMFF	3059	1088	106	39	19	12
LMF	2629	943	108	54	29	22
SMNF	2541	966	101	35	19	12
SPTF	2530	968	95	37	21	13
SJF	1898	712	86	33	20	12

(b) $\bar{q}$ Versus p

Table 11. Comparison of Eight Different Scheduling Algorithms.

loaded, all the jobs will be served according to their arriving order and no scheduling algorithm will be used. When the system load exceeds a certain threshold, a scheduling algorithm, e.g., SJF, will become effective to schedule the jobs waiting in the queue.

In Figure 20, we have shown the sensitivity of the turnaround time when we increase the number of processors. Table 11 also shows the same phenomenon. Moreover, the drop is even sharper here. This again shows the importance of having enough processors in the system.

As we said at the beginning of this section, the scheduling algorithm is used to determine the order that the waiting jobs will be considered for entering the memory. In general, all the jobs will be lined up in the queue according to the scheduling algorithm. Hence, the job at the head of the queue will always be considered first. Of course, this job might not be able to get into the memory when we are looking for a job to be executed. A new problem arises here, that is, shall we skip this job and consider the next one? This is the so-called "look-ahead" problem.

Apparently, there is no reason we should not consider the second job if the first job gets blocked due to lack of memory, since we can shorten the queueing time of the second job if it can fit into the memory. So, it is conceivable we might improve the average turnaround by doing look-ahead.

Naturally, the next question is, if we allow looking ahead, do we consider the third job if the second one still cannot enter the memory. In other words, do we allow look-ahead to be carried onto the third job. This is usually called the "look-ahead distance" problem. The look-ahead distance is defined to be the maximum number of jobs we can look at down the queue.

For example, if the look-ahead distance is 2, then we can at most look at the first two jobs and cannot look beyond the second job. Therefore, the look-ahead distance of one is equivalent to the non-look-ahead.

It is true that the chance of finding a "fittable" job is better if we allow longer look-ahead distance. But, it does not necessarily mean we can get a better average turnaround time by increasing the look-ahead distance, since the original order set up by the scheduling algorithm will be perturbed by the look-ahead scheme, and the longer the distance is the more this order will be perturbed. In other words, the look-ahead distance will cancel the effect of the scheduling algorithm. So, a large look-ahead distance might not be desirable.

Figure 27 shows the effect of look-ahead. We can see when we allow a moderate look-ahead distance, say 4, we do gain some benefit. However, a larger look-ahead distance might even cause a negative effect! Therefore, we suggest a look-ahead scheme with a moderate distance.

3.1.5 Effects of Job Characteristics

In this section, we are going to study the effects of job characteristics. As we mentioned earlier, each job is characterized by four parameters, namely, the arrival time, the CPU time, the job size, and the number of I/O requests. Needless to say, everyone of them will affect the system performance. Our purpose here is to find out how sensitive the effect of each parameter is.

Let us first look at the arrival time. The arrival time determines how fast the job stream puts work on the system. Of course, the faster the jobs arrive, the heavier the workload will be. Since the processing power

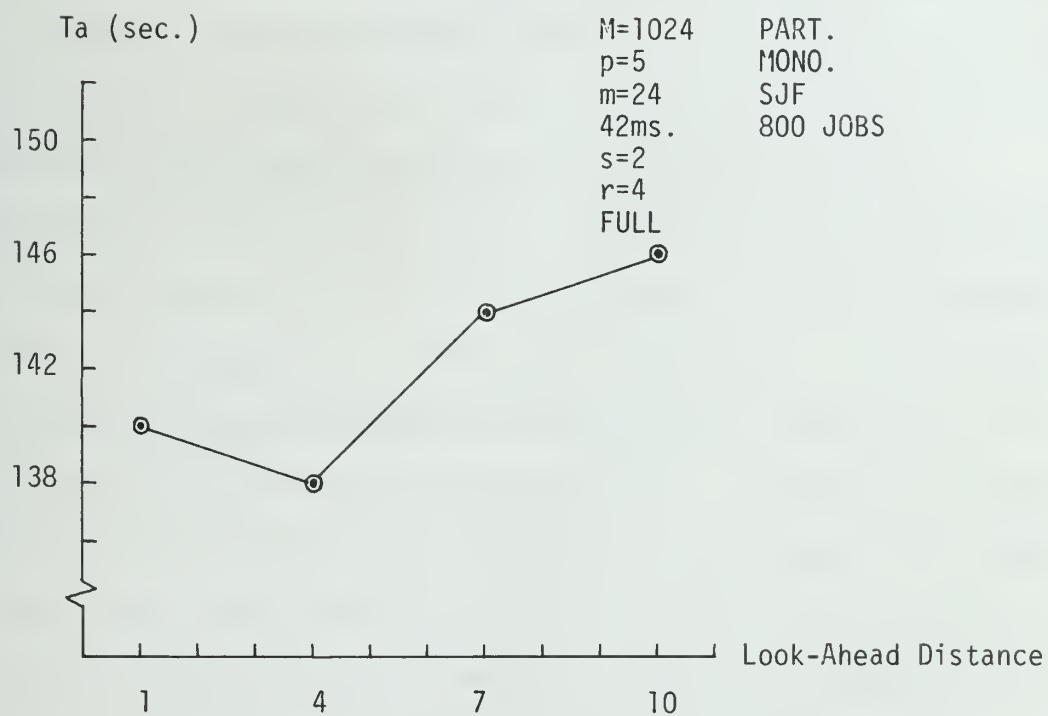


Figure 27. The Effect of Look-Ahead Scheme.

and the memory space are limited, more jobs will be accumulated in the waiting queue if the jobs come in faster, and thus each job will suffer longer delay. Apparently, the average turnaround time should go up as we increase the arrival rate.

In order to change the arrival rate, we will multiply the arrival time of each job by a variable called the "Arrival Scaling Factor." We can speed up the arrival rate by using a smaller arrival scaling factor, since the arrival time of each job will then be scaled down to a smaller value.

Figure 28 shows how the average turnaround time responds when we change the arrival scaling factor. As we can see, when we decrease the scaling factor from 1.0 down to 0.3, the average turnaround time does not change very much. Apparently, the system is unsaturated, or lightly loaded, within this range. In other words, the jobs do not arrive as fast as the processors can process. This does not mean our IBM 360/75 system is under-loaded, even though the job characteristics are obtained from analyzing the real workload on this machine. It is because we are using more processors in our model, and hence our system has a higher processing power.

However, when we further decrease the arrival scaling factor, T_a starts going up. Especially beyond 0.1, T_a increases very sharply. Obviously, the system starts getting saturated at around 0.1. Heavier loading will push the system into oversaturation, and the average turnaround time starts to blow up.

When the jobs arrive faster, it is conceivable the system will become busier. This is because the possible idle periods will become smaller. Figure 29 shows how the percentage of the time the system is busy increases as we decrease the arrival scaling factor. By busy, we mean at

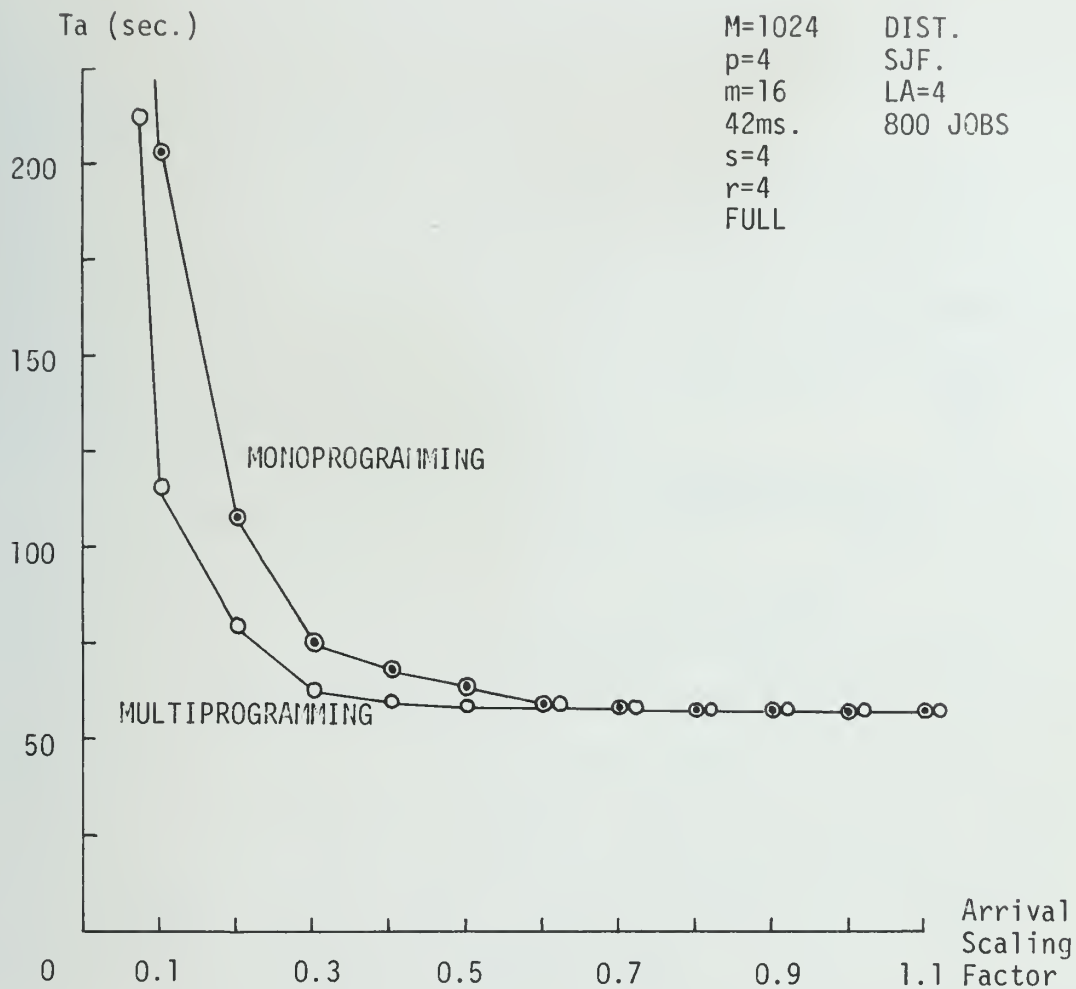


Figure 28. The Effect of the Arrival Scaling Factor on the Average Turnaround Time.

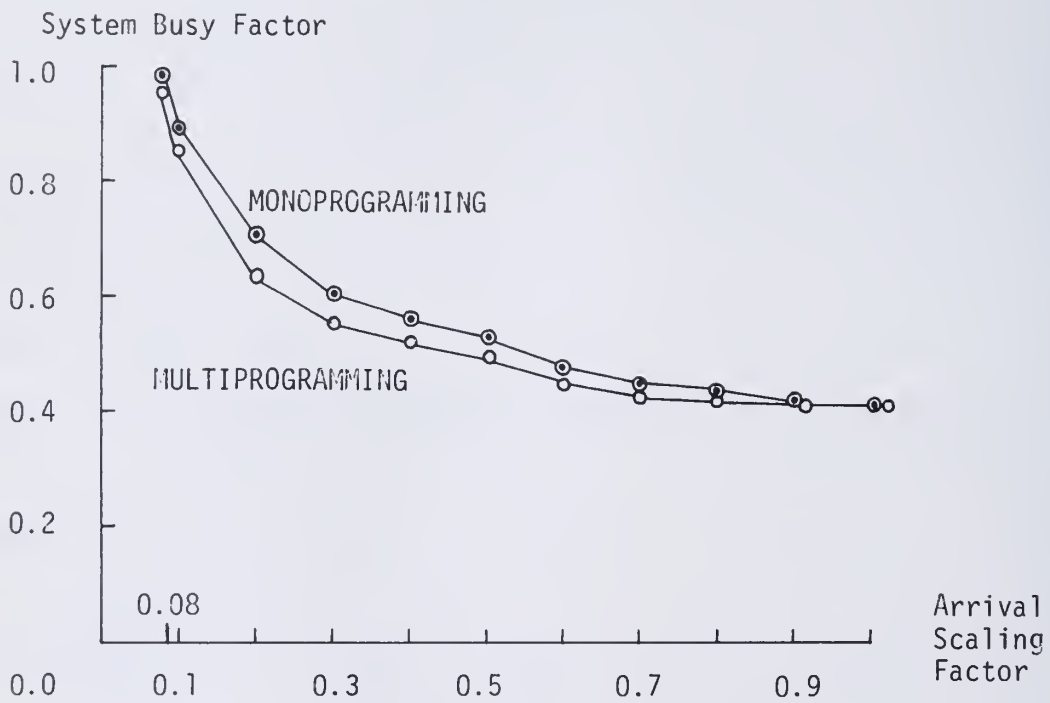


Figure 29. The Effect of the Arrival Scaling Factor on the System Busy Factor.

least one job is in the memory, no matter whether it is doing I/O or being processed by a processor. At 0.08, the system is busy for more than 95% of the time. This proves that the system is indeed getting saturated when we decrease the scaling factor below 0.1.

When the system is in saturation, measures of throughput or turnaround time might not reflect the true effects of some system parameters, and similarly when the system is far below saturation. Therefore, we scale the arrival time by a factor of 0.1 in all our simulations. This places the system in an interesting region.

Now, let us look at the effect of the job size distribution. As we said earlier, one major reason that a job gets blocked from being executed is due to the lack of memory. If we fix the memory size, apparently the job size will have a big impact on the system performance. Of course, the larger the job sizes are, the smaller the average number of jobs the memory can contain is, and the more frequently a job will get blocked. Therefore, we should expect to get a longer average turnaround time if we use a job mix which has a larger job size distribution.

Figure 30 shows the effect of the job size on the average job turnaround time. We fix all other parameters and change the job size by multiplying the size of each job with a Job Size Scaling Factor. This is similar to what we have done on the job arrival time. Thus, if the scaling factor is 2, the size of each job will be doubled. We can see the turnaround time is very sensitive to the change of the job size. When we double the scaling factor from 1.0 to 2.0, the average turnaround time increases by almost 150% (from 83 to 209). Again, T_a doubles when we increase the scaling factor from 2.0 to 2.5. Apparently, T_a grows exponentially when we increase the

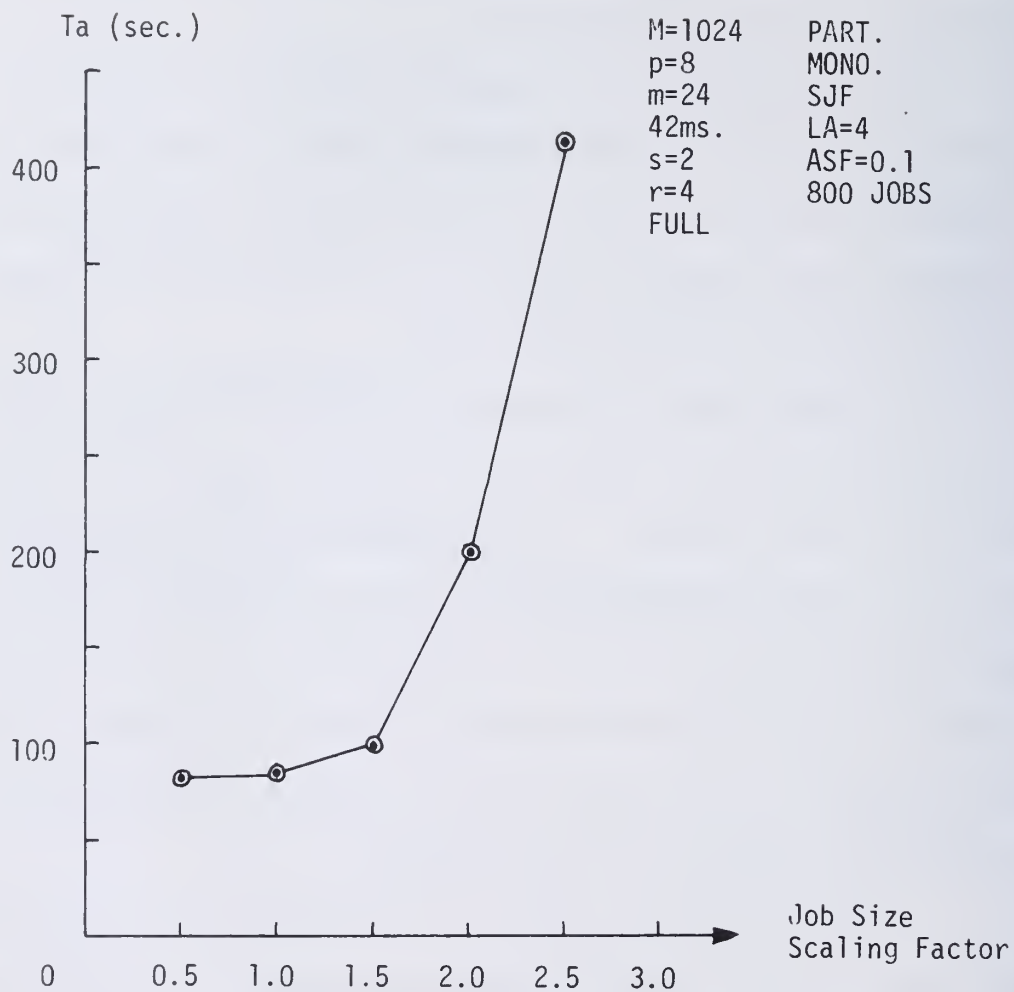


Figure 30. The Effect of the Job Size on the Average Turnaround Time.

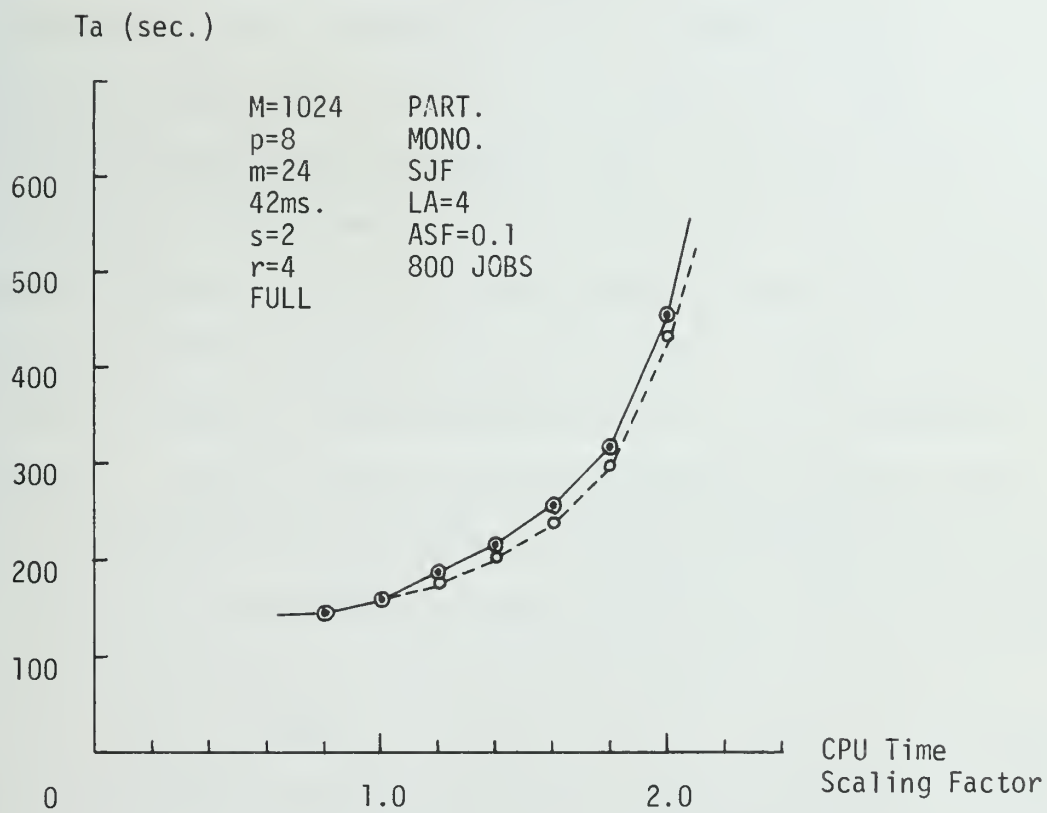


Figure 31. The Effect of Increasing CPU Time.

job size. This indicates the importance of having enough memory.

Table 12 shows the corresponding memory utilization. Just as we have expected, the memory utilization increases when the job size goes up. This again tells us the memory utilization should not be used as an indication of the performance.

The limited size of memory is usually the bottleneck in a computer system. Figure 30 shows that this is especially true in a multiprocessor system. However, this does not say the use of an extremely large memory will always do some good. When we reduce the job size scaling factor down to 0.5, this is equivalent to using a memory which is twice as large, we still get the same performance, but the memory utilization has been decreased to only 42%. This means we are wasting the memory and getting no improvement at all. So, an appropriate amount of memory should be used in order to get both good performance and good utilization. From Table 19, we can see 1024K bytes is the best memory size for the workload we are using. This is why we use this memory size throughout this chapter.

Finally, let us look at the effect of increasing CPU time or the number of I/O requests. Both of them will increase the time a job has to spend in the memory. This in turn will increase the time those waiting jobs have to spend in the queue. Therefore, the increase of either CPU time or the number of I/O requests of a job has a twofold effect on the average turnaround time. We briefly mentioned this in Section 3.1.2 when we compared monoprogramming and multiprogramming.

We will discuss the effect of increasing CPU time first. Again, we use a CPU Time Scaling Factor to scale the processing time of each job. Figure 31 shows how the average turnaround time responds when we increase

Job Size Scaling Factor	Ta	U _m
0.5	84	0.42
1.0	83	0.58
1.5	96	0.76
2.0	209	0.87
2.5	402	0.88

Table 12. The Average Turnaround Times and Memory Utilizations for Different Job Sizes.

(or decrease) the CPU time. The curve first grows slightly more than linearly, and then starts taking off if we double the CPU time scaling factor. This is not surprising since the system is getting saturated when we scale the CPU time beyond 2.0.

The use of scaling factor effectively "stretches out" the distribution curve since every job is enlarged by the same factor. Theoretically, if we use a random number generator with the same distribution and the scaled mean to generate a job stream, these jobs should have a distribution roughly the same as the stretched distribution. In other words, two methods should produce the same characteristics. The dotted curve in Figure 21 displays the result by using the random number generator method. We can see there is not much difference between these two methods. Therefore, we use the scaling factor method since it is easier to apply.

The curves of Figure 31 are very similar to that of Figure 30. In fact, they almost coincide with each other. Apparently, both CPU time and job size have the same effect on the system performance. One other interesting point is, all these curves are very similar to the turnaround time curve of an M/M/1 queueing system. The latter can be expressed by the following equation:

$$T = \frac{1/\mu}{1-p} = \frac{1}{\mu-\lambda}$$

where μ is the service rate, λ is the arrival rate, and $p = \lambda/\mu$ is the system utilization. Increasing the CPU time or the size of a job is in fact equivalent to reducing the service rate. T increases when we decrease μ . When μ gets very close to λ , T will become extremely large since the system is about to saturate. This can explain why the T_a curve goes up very sharply

when we scale the CPU time beyond a certain limit.

Table 13 shows the average turnaround, queueing and service times of a job when we use a different scaling factor. As we can see, most of the increment comes from the average queueing time. This is what we should expect, since each job will remain in the memory for a longer time but the arrival rate remains the same, hence more jobs will be accumulated in the waiting queue and a new arrival will have to spend much more time in the queue.

When we design a system, we should be very careful about the job arrival rate, the average CPU time, and the processing power we have. If the average turnaround time falls on the high rising edge of the curve, we should try to lower it by adding more hardware.

3.2 Results for Hardware Related Questions

In the second half of this chapter, we are going to answer the hardware questions we listed in Chapter 1. Mainly, we will investigate the effects of using different amounts and different speeds of hardware, then find a cost-effective way of building a machine.

We will also look into a very interesting problem, namely, the processor-memory inter-connection problem. The results we have shown so far are using a full processor-memory connection scheme, i.e., each processor is physically connected to every memory module and can access any module. For example, a crossbar switch is a full connection scheme. Under this scheme, each module can be assigned to any processor. Jobs cannot be prevented from entering the system due to the inability to connect available memory to an available processor.

CPU Time Scaling Factor	T_a	$\bar{q}$	$\bar{x}$
0.8	74	11	63
1.0	83	12	71
1.2	94	19	75
1.4	108	29	79
1.6	128	41	87
1.8	155	62	93
2.0	230	131	99

Table 13. The Effect of Increasing CPU Time on the Average Queueing Time ($\bar{q}$) and Average Service Time ($\bar{x}$).

However, as we mentioned in Chapter 1, a full connection is very expensive. Its cost can go up very quickly if we want to expand the system. For example, if we want to double a 4x8 system to a 8x16 system, the cost of the connection network will go up four times. Besides, a full connection network usually is not easy to expand. For example, in a crossbar switch, it is very difficult to increase the size of the fan-out tree, since that requires a complete rewiring between the fan-out tree and fan-in tree. Or, in a multiport memory system, we will have to replace all the memory interfaces if the number of processors of the expanded system exceeds the number of ports in a single module.

Therefore, we are interested in using a partial connection network. Let us recall the connection network of the PRIME system shown in Figure 1. Each processor is connected to 8 memory modules via a private bus. Each memory module has four ports, so up to four processors can connect to one module. In the current PRIME system, there are five processors and 13 memory modules, so 40 out of the 52 ports are being used. This is a typical partial connection system. In our study of partial connection, we will assume this kind of architecture.

Of course, we would expect some degree of performance degradation, since each processor only connects to a subset of the modules and thus can only access part of the memory. Hence, a processor cannot be assigned to a job if the memory attached to this processor is not big enough, even though the total available space in the memory is big enough. Consequently, the probability that a job will be blocked is larger in a partial connection system. However, the cost of a partial connection does not grow as fast as a full connection. In fact, it will grow linearly if we use multiport memories.

Most of all, the partial connection does allow us to expand the system without too much trouble. If we are increasing the memory, we can just connect the additional module to some processor arbitrarily or following some rule. If we are also increasing the number of processors, we might need to move some connectors to reconfigure the whole system. But, no hardware modification is needed.

We will study the performance degradation of a partial connection network. In addition, we will look at an interesting problem, namely, how should we interconnect the processors and the memories such that we can get a minimal degradation. In general, a processor will be allowed to connect to only part of the memory in a partial connection system. How many modules should be assigned to a particular processor will greatly influence the job handling ability of that processor. Obviously, the more memory a processor connects to, the larger job it can handle.

In the PRIME system, all processors are connected to the same number of memory modules, namely, 8. Hence, all of them are equally "capable." Of course, this is based on the assumption that no job will ever require more than eight modules, and the fact that there are enough ports in the memory. Obviously, if any of these two conditions are violated, this connection will not work any more, and we need a new configuration. For example, if some jobs need 12 memory modules, then at least one processor should be connected to that many modules. However, not all of them can have 12 memory modules since for 5 processors that would require a total of 60 memory ports. Therefore, an "uneven" connection might be more effective, that is, some processor will have more memory and some will have less.

How to distribute the total available ports to the processors in this case is a very interesting distribution problem. This is part of the interconnection problem we will be looking at later.

After we determine the number of memory modules each processor will get, another problem arises immediately, namely, which module we should choose to connect to a certain processor. In the PRIME system, this is done in a rather uniform way. Each processor will connect to 8 consecutive modules with the leading module two modules to the right of the leading module of the previous processor. Of course, this might not be a good way to arrange the connection. However, it is very difficult to come out with an appropriate analytic argument to show what a good connection might be. What we will try to do is to simulate several combinations and compare their results. Then, analyzing the connections which yield better results we can get some ideas of what we might need to do in order to achieve a good connection. This is the second part of the interconnection problem we will be studying.

3.2.1 Hardware Quantity Effect

Let us now look at how the system responds when we increase the hardware. Of course, the more hardware we add into the system, the better performance we should get. What we want to know here is how sensitively each type of hardware resource will affect the system performance. Therefore, we shall know what to buy in order to achieve a certain percentage of improvement and spend as little money as possible.

In our system, there are four kinds of hardware we can increase: the total amount of memory M , the number of processors p , the number of memory modules m , and the number of I/O devices r . We will investigate the effect

of each individual parameter. So, when we change a certain parameter we will assume the other three are fixed. At the end of this section, we will also show what will happen when we increase them simultaneously.

The effect of increasing the total amount of memory M , keeping p , m , and r fixed, is actually the same as that of decreasing the size of each job. This is because both actions have the same effect of allowing more jobs to enter the memory and get executed at the same time. Obviously, doubling the memory M is equivalent to halving the size of each job. Therefore, changing M should yield the same result as that shown in Figure 30, except if we plot T_a against increasing M the curve will become exponentially decreasing instead of increasing. Our simulation indeed shows exactly the same numerical result and so we omit the repetition.

The effect of increasing the number of processors p has actually been shown in Figures 18 and 20. In Figure 32, we repeat this for three different m values. What we want to do here is to compare the effects of increasing p and m . In Figure 33, we use the same values of T_a and plot the curves against m .

From Figure 32, we can see how important it is to have enough processors. In this case, at least five processors should be used in order to get good performance. We have stressed the same point earlier in this chapter. In Table 14, we show both the memory utilization and the average memory bandwidth for each job. We can see in Table 14-b the job memory bandwidth, i.e., the average memory bandwidth each job gets, does not change when we increase p . This is what we should expect since we are dealing with a monoprogrammed system. So, the performance improvement must come from the increase of memory utilization alone. In Table 14-a, we can see the

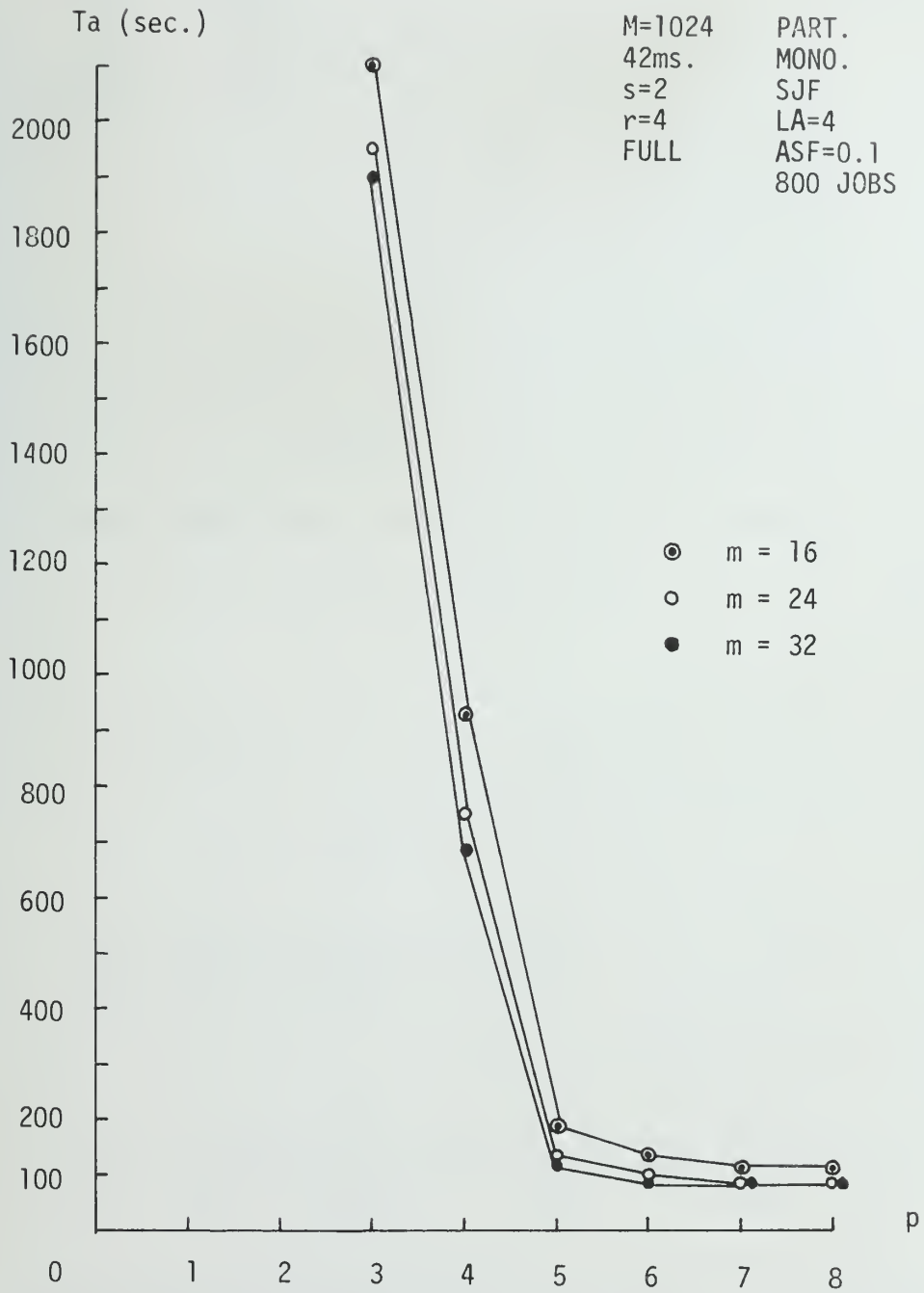


Figure 32. The Effect of p on the Average Turnaround Time T_a .

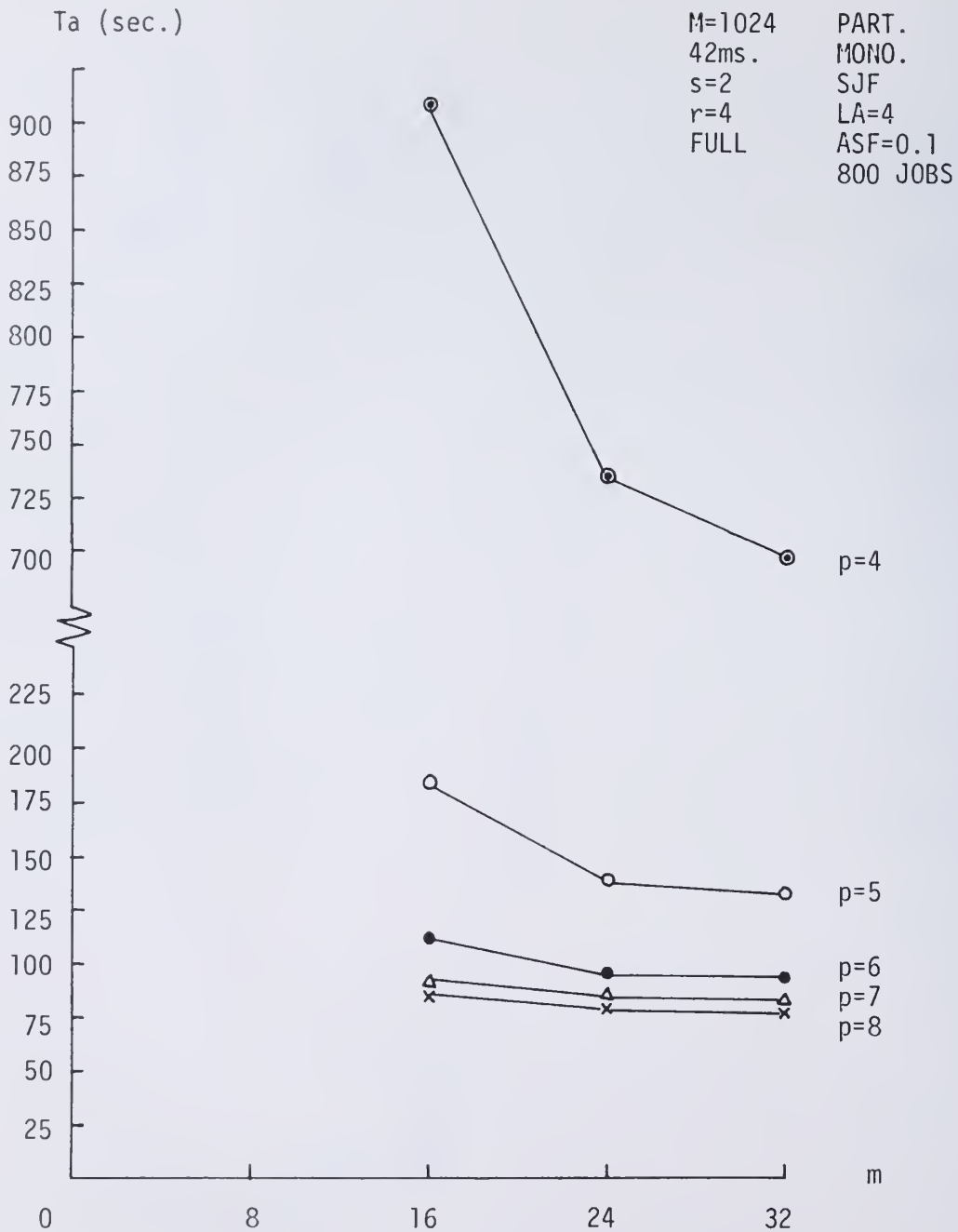


Figure 33. The Effect of m on the Average Turnaround Time T_a .

$\begin{array}{c} m \\ p \end{array}$	16	24	32
4	41.3, 14.1	41.3, 9.5	41.4, 7.0
5	51.5, 17.6	50.0, 11.5	49.1, 8.3
6	53.0, 18.0	51.8, 11.8	51.1, 8.7
7	53.4, 18.4	52.4, 12.0	51.6, 8.7
8	54.2, 18.5	53.1, 12.2	52.2, 8.9

(a) Percentages of Memory Utilization and Memory Waste of the Partitioned Scheme

$\begin{array}{c} m \\ p \end{array}$	16	24	32
4	1.35	1.45	1.52
5	1.36	1.46	1.52
6	1.36	1.46	1.52
7	1.36	1.46	1.52
8	1.36	1.46	1.52

(b) Job Memory Bandwidth

Table 14. The Memory Utilization, Memory Waste, and Job Memory Bandwidth for the System of Figures 32 and 33.

		$(B_w^T , \bar{q})$		
p	m	16	24	32
4		2.69, 852	2.78, 708	2.84, 629
5		3.27, 114	3.28, 74	3.29, 67
6		3.30, 41	3.31, 30	3.31, 29
7		3.31, 25	3.32, 17	3.33, 16
8		3.32, 23	3.33, 15	3.33, 14

Table 15. The Total Memory Bandwidth (B_w^T) and Average Queueing Time ($\bar{q}$) for the System of Figures 32 and 33.

memory utilization increase by as much as 13% as we double the processors from 4 to 8. When $p=4$, the memory is indeed under-utilized. Of course, this is because we only allow up to four jobs in the memory so a lot of jobs have to wait in the outside queue. We can see from Table 15 that the queueing time for $p=4$ is very large. When we increase p , we actually allow more jobs to be in the memory at the same time, and hence the memory utilization goes up, and the queueing time goes down. Meanwhile, the average service time increases due to the competition of I/O devices. For large p , say 7 or 8, the curves become flat, and no improvement will result even if we add more processors. From our simulation result, we see that most of the time the memory can only contain six to seven jobs. So, any more processors beyond that will simply be wasted.

From Figure 33, we can see the turnaround time also goes down when we increase m . In fact, the increase of m has two fold effect on the system performance. It can reduce the memory waste since the module size will be reduced. (Notice that, when we increase m , we are holding the total amount of memory fixed.) This can be seen in Table 14-a. Also, it can increase the memory bandwidth since the degree of interleaving for each job will be increased. This can be seen in Tables 14-b and 15. Since the speed ratio s is only 2, the increase of m will only cause small improvements on the total memory bandwidth and the job memory bandwidth.

When we increase m from 16 to 24, a significant improvement has been achieved. This is because the memory waste has been reduced significantly, and the job memory bandwidth has been increased by a non-trivial percentage (about 8% in this case). When we again increase m from 24 to 32, a small change has been made on the turnaround time except when p is small. Therefore,

an m value of 24 should be enough to achieve good performance. This phenomenon can also be seen in Figure 19 and 22.

In Table 14-a, we can see the memory utilization actually decrease when we increase m . This is because the throughput has been increased, and a job will stay in the memory for a shorter period of time. We have explained this when we discuss Table 7.

From Figures 32 and 33, we can see that the number of processors has the most profound effect on the system performance. If we do not have enough processing power, the increase of any other hardware will turn out to be wasteful.

Figure 34 shows the effect of another parameter r , the number of I/O devices. Here we use 8 processors, 24 memory modules, and the partitioned scheme. As we can see, the breaking points occur at $r=4$. When we increase the number of I/O devices from 2 to 4, the average turnaround time improves significantly. This can be explained by Table 16. Table 16 shows the average queueing time of a job waiting for an I/O device. For $r=2$, the queueing time is very large. Apparently, many jobs jam up in the I/O queue due to lack of I/O channels. When we double the number of I/O devices to 4, the queueing time drops drastically. This can be seen in the first two rows in Table 16. The reason is rather simple: since we are using monoprogramming and 8 processors, at most 8 jobs will be in the memory simultaneously, and since the jobs are not particularly I/O-bound, the probability that more than four jobs are doing I/O is small. Hence, four I/O channels will be enough in this case. Further increase of I/O channels gives very little improvement to the performance. In fact, this is also true for the multiprogramming case, except the multiprogramming case shows a little bit higher queueing time since on

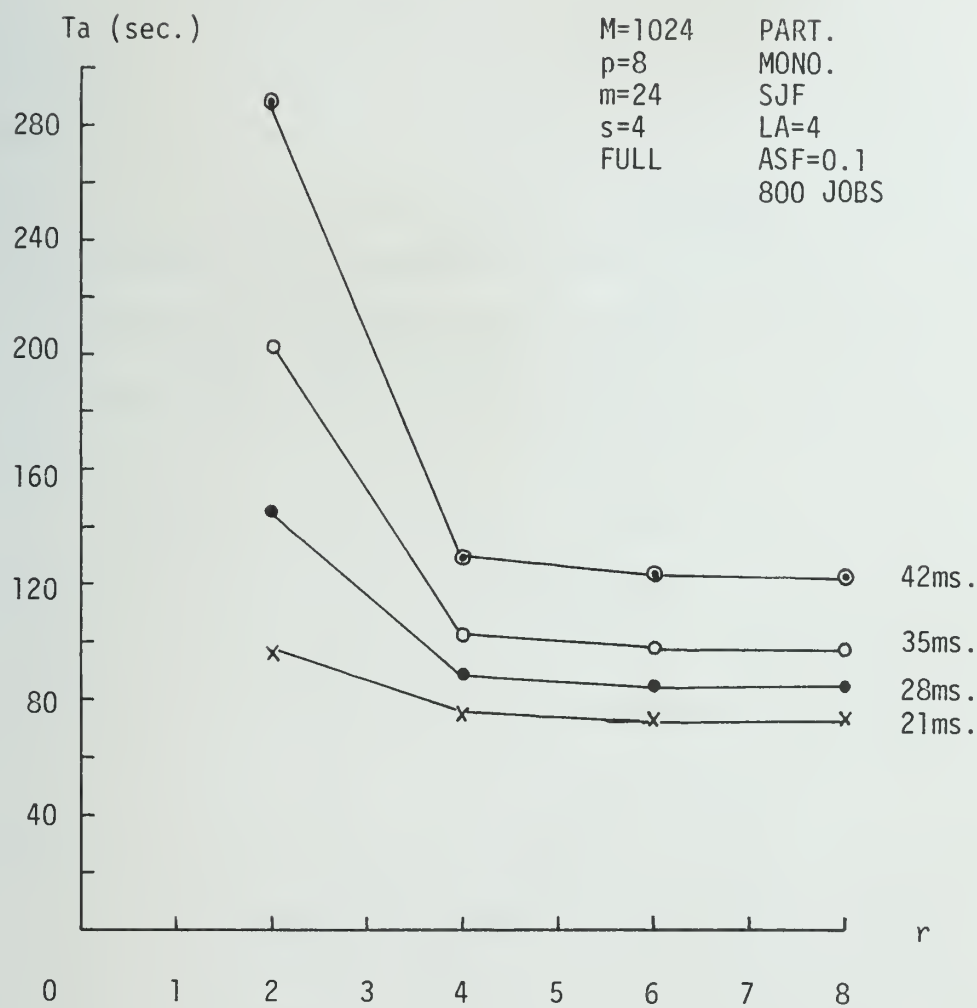


Figure 34. The Effect of the Number of I/O Devices.

r \ Unit I/O Time(ms)	21	28	35	42
2	10.2	23.3	40.5	53.6
4	0.9	1.9	3.6	6.6
6	0.9	1.8	3.5	6.3
8	0.9	1.8	3.5	6.3

Table 16. The Queueing Time for I/O Device.

the average more jobs will be competing for the I/O channels. Of course, if we are dealing with an I/O-bound job mix, more I/O devices will be needed then, since the I/O stage will become the bottleneck of the system.

From Figure 34, we can also see that the percent difference of the turnaround time between $r=2$ and $r=4$ decreases as we reduce the average time per I/O operation, i.e., the curve becomes flatter for faster I/O devices. This is because the average unit I/O time has a twofold effect on the turnaround time. When we reduce the average unit I/O time, both the job I/O time and the queueing time for I/O has been reduced at the same time. In particular, the queueing time drops by a rather large factor when the average unit I/O time is reduced. This can be seen in each row of Table 16.

One more interesting thing is: the queueing time for $r=2$ with an average unit I/O time of 21 ms is almost 60% larger than that for $r=4$ with an average unit I/O time of 42 ms (Table 16). However, the average turnaround time for the former case is about 25% better than the average turnaround time for the latter (Figure 34). The first case can be explained as follows: Although both cases have the "same" power, using four slow I/O devices a job will not suffer any delay unless there are already four or more jobs doing I/O. In other words, a job has a lower probability of being enqueued. But, using a faster I/O device can reduce the total job I/O time. Apparently, in these two cases what we gain from the total I/O time is much more than what we lose in the queueing time. This implies that if the cost of an I/O device with an average I/O time of 21 ms (e.g., IBM 3330 disk unit) is no more than twice of the cost of an I/O device with an average I/O time of 42 ms (e.g., IBM 2314 disk unit), then it might be a better idea to use half as many of the faster one. We will further look at the effect of the I/O speed in the next section.

Now, let us look at how the system performance reacts when we simultaneously increase the number of the processors, memory modules, and I/O devices. Of course, we would expect the system performance, e.g., the average turnaround time, to be improved at a much larger rate as we increase them at the same time. But, in order to make a fair comparison of the "capability" of each system, we will also adjust the workload of each system by scaling the job arrival time. For example, if we double the size of a certain system, we will also double the system workload by doubling the job arrival rate.

Figure 35 shows how the average turnaround time reacts when we double the system size from (4,12,2) to (8,24,4), then to (16,48,8). Of course, we double the job arrival rate everytime we double the system size. Notice that in these experiments, when we double the number of modules we also double the total memory size since we keep the module size the same. In the (4,12,2) system, we use 512K bytes of main memory. So, we use 1024K and 2048K bytes in (8,24,4) and (16,48,8) systems respectively.

As we can see, the Ta curves (solid curves) drop roughly exponentially as we double the system size, even though we double the job arrival rate at the same time. This means that if we double the system size, we should be able to handle more than twice the workload. In Table 17, we show the total memory bandwidth, the job memory bandwidth, the average number of jobs in the system, and the queueing time for three different system sizes. As we can see, the total memory bandwidth increases very rapidly when we double the system size. This is the major reason that the average turnaround time improves so quickly since the total memory bandwidth is the amount of work being done in a unit of time. Only the job memory bandwidth

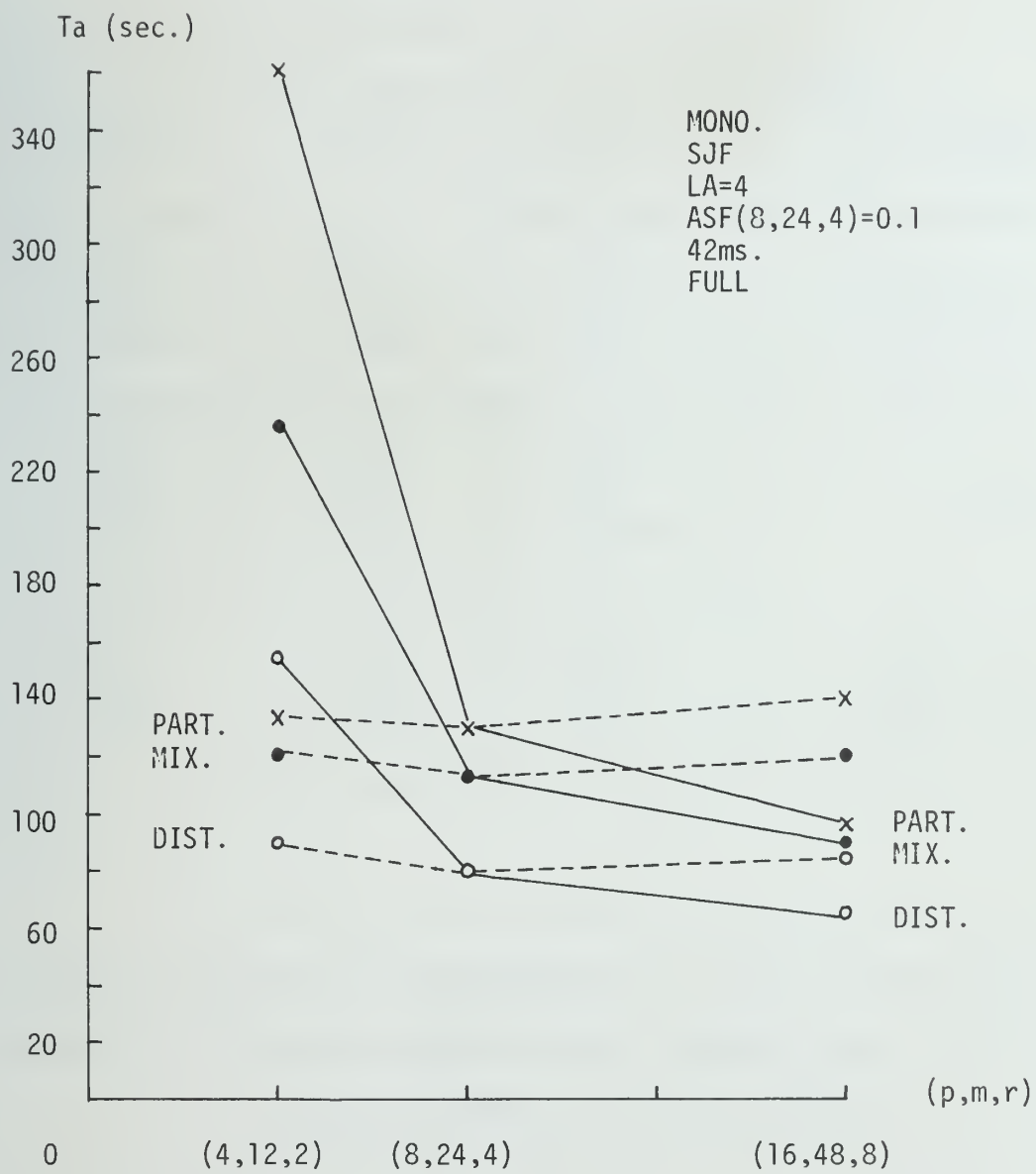


Figure 35. The Average Turnaround Time for Different System Size.

Allocation Scheme \ System (p,m,r)	(4,12,2)	(8,24,4)	(16,48,8)
Distributed	3.57 2.67 3.04 65.8	6.66 2.79 5.19 11.1	9.97 2.88 9.35 1.4
Mixed	3.39 1.75 3.50 141.6	6.60 1.75 6.46 27.9	9.91 1.76 12.16 4.2
Partitioned	3.36 1.75 3.38 266.6	6.57 1.75 6.40 37.1	9.89 1.75 12.09 5.8

Table 17. The Total Memory Bandwidth (B_w^T), Job Memory Bandwidth (B_w), Average Number of Jobs in System ($\bar{n}$), and Average Queueing Time ($\bar{q}$) for each System Size.

of the distributed system increases when we enlarge the system size. This is because the degree of interleaving has been doubled which reduces the memory interference between jobs. But, the important thing is the average number of jobs in every system almost doubles every time we double the system size, which causes the large increase of the total memory bandwidth. This implies the doubled system has twice the capability of containing jobs in memory. Of course, this seems rather intuitive since we also double the memory. However, in a smaller system, it is very possible that a few large jobs will occupy the memory and block the other jobs for a long time. But, this is less likely to happen in a larger system since it is unlikely several very large jobs will compete the memory at the same time. In other words, a larger system will have a higher potential of allowing more jobs to enter the memory. So, a job will pass through the large system quicker than the small system. In queueing theory, this is called the diminishing effect. In Table 17, we can see the queueing time decreases very fast when we increase the system size.

In fact, we can show that a double-sized system can handle work about 2.8, or roughly $2^{1.5}$, times the workload. This is shown by the dotted curves in Figure 35. The dotted curves are obtained by the following method.

Let us fix the performance of the (8,24,4) system and try to bring the performance of the other two systems close to it by adjusting the arrival rate. In order to lower the turnaround time of the (4,12,2) system, we slow down the arrival rate by increasing the arrival scaling factor. Notice that the larger the arrival scaling factor is, the lower the arrival rate will be. In Figure 35, we found that if we can use an arrival scaling factor of 0.28 ($\approx 0.1 \times 2^{1.5}$) the (4,12,4) system will

perform roughly the same as the (8,24,4) system. In other words, we make the arrival rate, and hence the workload, of the (4,12,2) system 2.8 times smaller than the (8,24,4) system and get almost the same turnaround time. On the other hand, we use an arrival scaling factor of 0.039 ($\approx 0.1 \div 2^{1.5}$) for the (16,48,8) system and the turnaround time increases a little bit to the neighborhood of that of the (8,24,4) system.

Therefore, all three systems now perform more or less the same but the workload ratio is kept roughly at $2^{1.5}$ between two consecutive systems which have a size ratio of 2. This means that the processing power of a double-sized system is about $2^{1.5}$ times larger than the original system. If we let c be the size of a system, we may say the processing power of our system grows roughly according to the function $c^{1.5}$. Since the cost of a system is directly proportional to the size of the system, we might as well think c as the cost. Therefore, the processing power P of our system can be formulated as follows:

$$P = a c^{1.5} = a c\sqrt{c},$$

where a is some proportionality constant. What this result implies is the performance will grow faster than linearly as we increase the size of the system.

However, we must point out that the above result only holds in the range shown in Figure 35. When we double the system size again to (32,96,16), the workload it can handle to yield a similar turnaround time does not grow by as much as 2.7 or 2.8 times of the workload of the (16,48,8) system. In fact, the (32,96,16) system can only handle a workload of about 2.3 times that of the (16,48,8) system. Obviously, arrival

rate has a larger effect on the performance than system size.

So, $c^{1.5}$ does not hold for system beyond (16,48,8). However, for a general system, (16,48,8) is already a reasonably large size. We can say that the above result is good for the range of system size most people will be interested in.

3.2.2 Hardware Speed Effect

In this section, we are going to study the effect of using faster components. There are two parameters we will look at, namely, the average unit I/O time and the memory-processor speed ratio. We did mention some effects of these two parameters earlier, however, we will look at this problem from a slightly different angle.

Let us first look at how the memory-processor speed ratio s will affect the system performance. For the convenience of this discussion, we will assume the processor speed to be fixed. So, the larger the s is, the slower the memory will be.

Figure 36 shows the T_a versus s curves for three different memory allocation schemes. As we can see, all three curves go up slightly more than linearly as we slow down the memory speed. However, the slope of the curve for the distributed system is smaller than those of the other two systems. This means that the memory speed has less effect on the distributed system. We can explain by using the bandwidth equation and the following example.

Assume we have eight memory modules, and three jobs in the memory which require 1, 3, and 4 modules respectively. Let us compare the partitioned system and the distributed system. If we use the random distribution assumption, we can apply Ravi's equation [38] to compute the memory

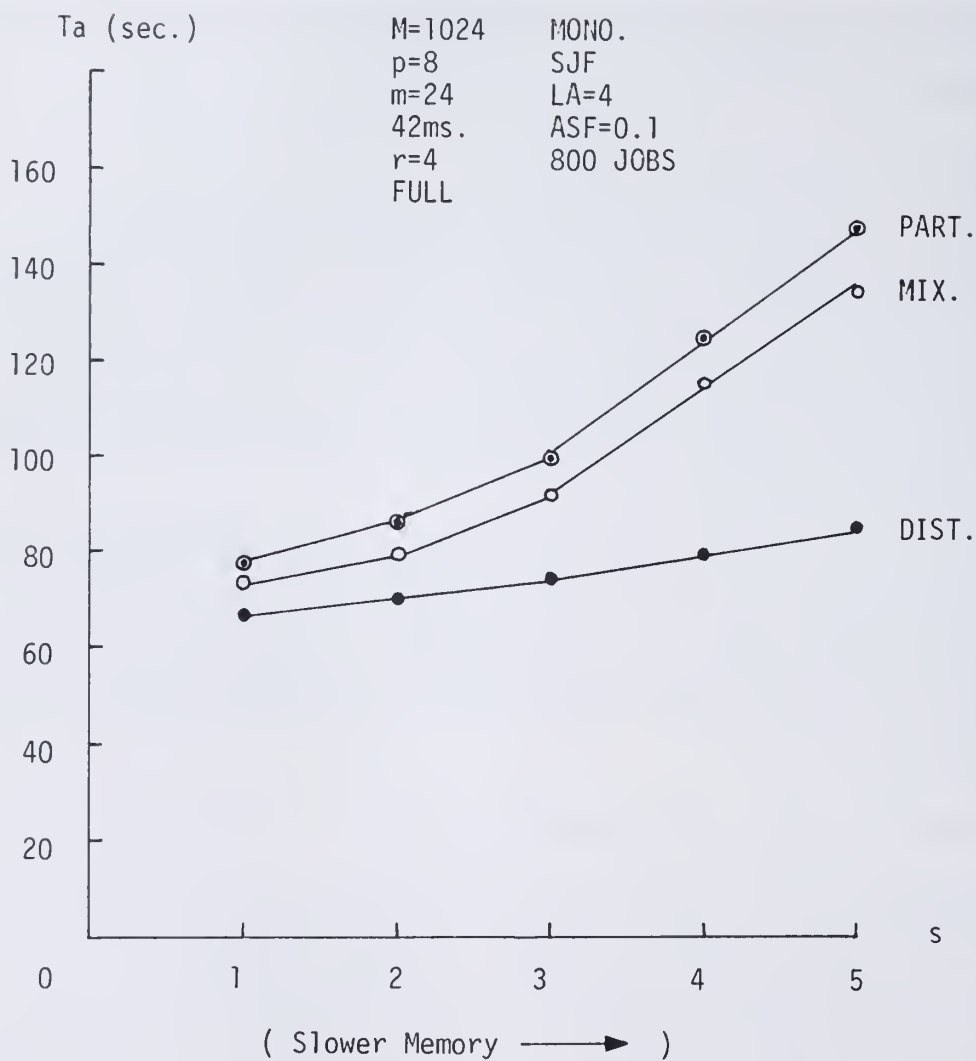


Figure 36. The Effect of the Memory-Processor Speed Ratio.

System	(Slower Memory $\longrightarrow$)				
	s=1	s=2	s=3	s=4	s=5
Distributed	2.64	4.41	5.60	6.39	6.92
	0.88	0.73	0.62	0.53	0.46
Partitioned	3.00	4.42	5.42	6.14	6.65
	1.00	0.73	0.60	0.51	0.44

Table 18. The Total Memory Bandwidths (Accesses per Memory Cycle) and the Number of Words a Processor can get per Processor Cycle for 5 Different s Values.

(3 Jobs in 8 Memory Modules)

bandwidth. For the distributed system, the bandwidth will be $8[1-(1 - 1/8)^{3s}]$, and for the partitioned system, it will be $1+3[1-(1 - 1/3)^s] + [1-(1 - 1/4)^s]$. We list their numerical values in Table 18. Notice that, although these values increase as s gets larger, the number of words a processor can fetch from the memory in a certain unit of time, say one micro-second or one processor cycle, is actually reduced. In Table 18, we also list the average number of words each processor can get in one processor cycle. This is done by dividing the bandwidth by $3s$, since the memory is s times slower than the processor and there are three jobs in the memory. We can see this "normalized bandwidth" is indeed decreasing when we increase s . This is because the memory cycle time is doubled when we double the speed ratio s , however, the values we get by using Ravi's equation do not double at the same time. Therefore, it in fact takes longer to fetch the same number of words out of the memory if s becomes larger. This is why the average turnaround time increases when we increase s . Moreover, as we can see from Table 18, the normalized bandwidth (number of words per processor cycle) of the partitioned system decreases faster than that of the distributed system. Thus, the average turnaround time of the partitioned system degrades faster than that of the distributed system. For the mixed system, the situation is very similar to the partitioned system except the turnaround time is a little bit better. This is, of course, because the mixed system has a rather similar way of storing the jobs but with a little bit better memory space utilization.

However, we can also look at Figure 36 from the opposite angle. When we increase the memory speed, or decrease s , the average turnaround time of the partitioned system improves faster than that of the other two

systems. For example, in this case the T_a of the partitioned system drops from 146 down to 83 as we reduce s from 5 to 2. That is a reduction of 43%. For the mixed system and the distributed system, the reductions are 39% and 12%, respectively. Therefore, the memory speed is a very important factor when we are using the partitioned scheme. After all, the partitioned system might yield a better bandwidth when the speed ratio is very small. We can see this, for example, in the $s=1$ column of Table 18.

Recently, the memory technology has provided system designers with faster, cheaper, and higher density semiconductor memories. It is now economically feasible to design a system which operates in the small speed ratio region. This makes the partitioned scheme more attractive to use, since it provides a very high reliability as well as competitive performance.

The second speed parameter we are going to look at is the average period of time spent on an I/O operation, or what we call the average unit I/O time. We have shown some effect of the average unit I/O time in the last section. Now, let us look at how this parameter affects the system performance.

Figure 37 shows the T_a curves versus the average unit I/O time for all three memory allocation schemes. All the curves have roughly the same increasing rate as we increase the average unit I/O time. This rate is larger than linear. We have explained that the reason is that this average unit I/O time has a two fold effect on the average turnaround time since it not only increases the total I/O time but also increases the queueing time indirectly. So, as we use slower I/O devices, the average turnaround time degrades rather quickly.

One figure we should pay some attention to is when we halve the

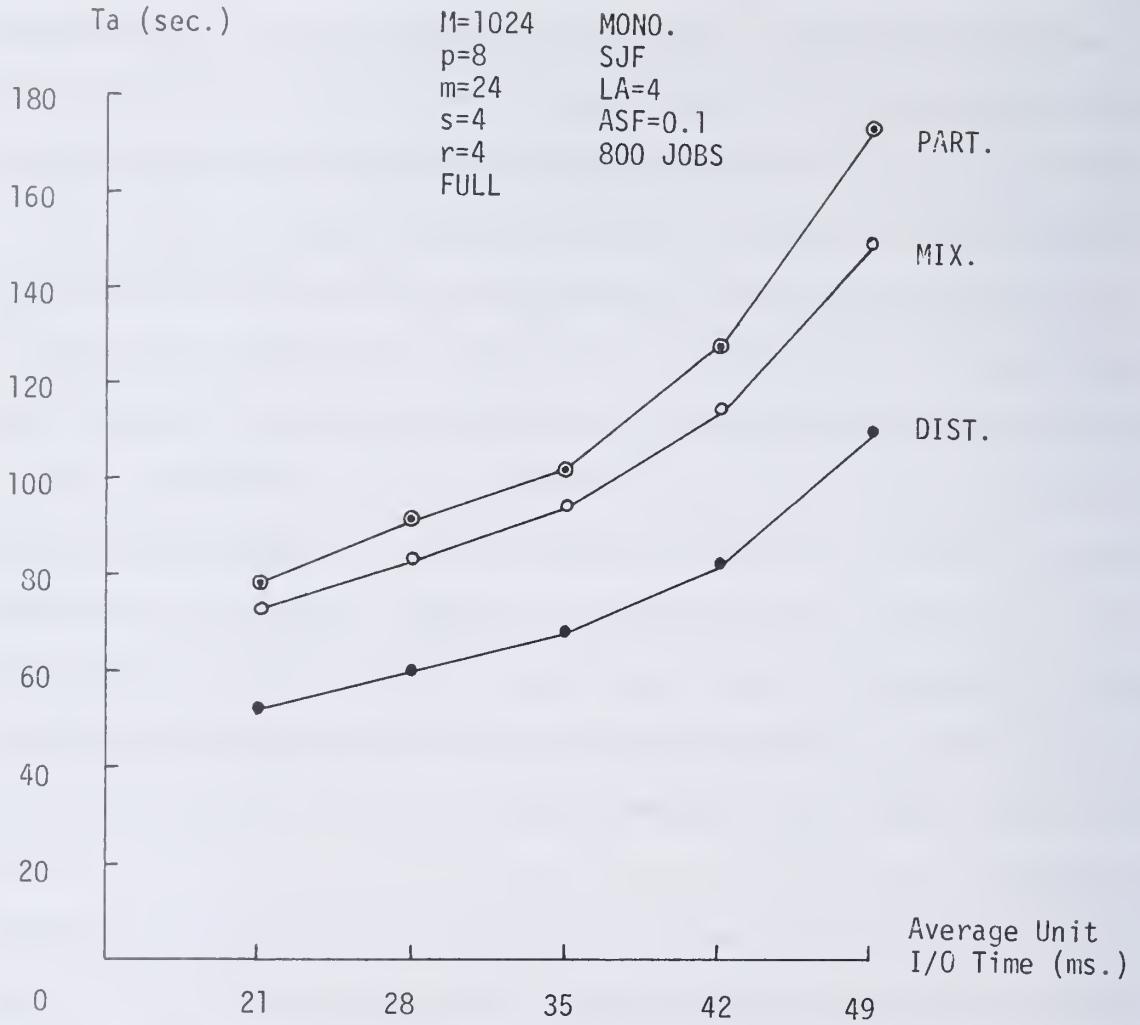


Figure 37. The Effect of the Average Unit I/O Time.

unit I/O time from 42 ms to 21 ms the average turnaround times all decrease by more than 40%. This is a larger effect than that of halving the memory cycle time. Therefore, using a faster I/O device will have a more significant improvement on the performance, at least given the I/O-execution time balance of our job load. Needless to say, the effect will be even larger if we are dealing with an I/O-bound job mix.

Of course, the type and numbers of I/O devices will depend on the cost and the resulting performance. For example, in Figure 34, we can see that using two 21 ms I/O devices can yield a better turnaround time than using four 42 ms I/O devices, and if the faster I/O device does not cost more than twice of the cost of the slower I/O device it is obvious that the faster I/O will be a better choice. However, it might be extremely expensive to replace a slower I/O device by a faster I/O device, since it may involve the replacement of the I/O controller and some very expensive equipment. So, it is very important to understand the effect of the I/O device before we can decide what to use in a system.

3.2.3 Partial Connection

In all the discussions we have up to this point, we assume our system to have a fully connected switching network (full connection in short), e.g., a crossbar network. As we described earlier, in this kind of system a processor is physically connected to all the memory modules and can access any module if it is allowed to do so. So, the operating system can freely assign any module to any processor as long as the resource management policy is not violated. However, the cost of a full connection will grow very quickly as we expand the size of the system. This is the

price we shall have to pay in order to maintain that availability.

Now, let us look at another kind of architecture which is a cheaper and more flexible alternative for interconnecting the processor and the memories, namely, the partial connection network. The best example of a partial connection is the multiport memory network used by the PRIME system, which we showed in Figure 1. We briefly talked about the advantages and the disadvantages of a partial connection network a few times in the early chapters. In this section, we are going to elaborate more about this subject.

Of course, the biggest (perhaps the only) disadvantage of the partial connection is performance degradation. The performance degrades when we reduce the connections between the processors and the memory modules. The main reason is that the utilization of the available memory has been seriously restricted by the partial connection. Very often, a job cannot be put into the memory because no processor has enough free space connected to it, although the total unused space is larger than what this job is requesting. This can be explained by a simple diagram. Figure 38 shows a partially connected system with three processors and six two-port memory modules which are interconnected in a uniform way similar to that in the PRIME system, and two jobs a and b are occupying four modules as shown in the figure. Suppose a third job arrives which requires two modules. It cannot enter the memory since the third processor has only one unoccupied module attached to it, although there are two free modules available in the system. Obviously, the memory will be wasted due to this incomplete interconnection. As a result, the system performance is degraded.

The memory waste caused by the partial connection is quite dif-

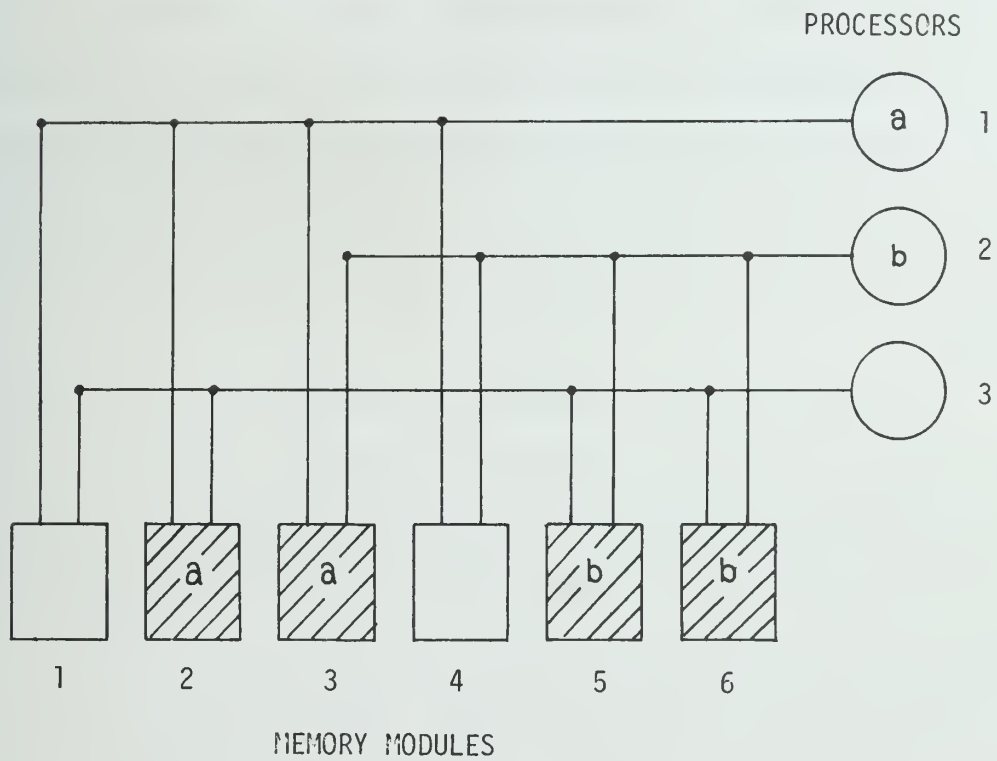


Figure 38. A Partial Connection Network.

ferent from that caused by the monoprogramming scheme. The memory waste here is created by the inaccessibility of a processor to a memory module. For example, in Figure 38, the fourth module is wasted since processor 3 does not connect to this module. Therefore, if the processors that connect to a certain module are all assigned jobs, then the unused portion of this module, probably the whole module, is wasted.

The memory waste highly depends on the number of processors attached to a module. If we reduce the number of processors that can access a module, the probability that part or all of this module will be wasted is increased. Of course, when the memory waste increases the system performance gets worse.

In our discussion of the partial connection, we will always assume all the ports of a module are connected to processors and none is left unused. So, the number of processors connected to a memory module is equivalent to the number of ports the module has. We will also assume that all the modules are identical with a number of ports which is no more than the total number of processors in the system. Since in a partially connected system all the processors might not connect to the same number of memory modules, we will use an array of integers to represent this system which can tell us how many modules a certain processor is connected to. For example, we will represent the system in Figure 38 by (4,4,4). In fact, the order of these integers is not important since the processor number is arbitrarily assigned. Notice that the sum of these integers is equal to the total number of ports in the memory. Of course, these numbers do not reveal the information about how the connections are being made. If the details of the connection are important to the discussion, we will use a method called

"connection matrix" to represent the connection. But in general, the connection will be very uniform as that in Figure 38.

The connection matrix is a succinct representation of a partial connection. The matrix has p rows and m columns with each entry indicating the connection between a processor and a memory module. If a connection is made between processor i and module j , we will put a 1 at the position (i,j) , otherwise, the entry will be 0. For example, the partial connection in Figure 38 can be represented by the following 3 by 6 connection matrix:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Notice that all the column sums are equal to the number of ports of a module, and each row sum is equal to the number of modules connected to the corresponding processor.

Figure 39 shows how the average turnaround time degrades when we reduce the number of ports of each memory module, or equivalently the number of processors connected to a memory module. When the number of ports per module is 8, it is equivalent to the full connection since we are using 8 processors. If we reduce the number of ports to 4, only half of the processors will connect to every module. Here, we use a (12,12,12,12,12,12,12,12) connection. The processors are connected to the memory modules in a uniform way. Each processor is connected to 12 consecutive modules with the leading module being skewed three modules to the right of the previous leading module. Using the connection matrix representation, this connection can be expressed by:

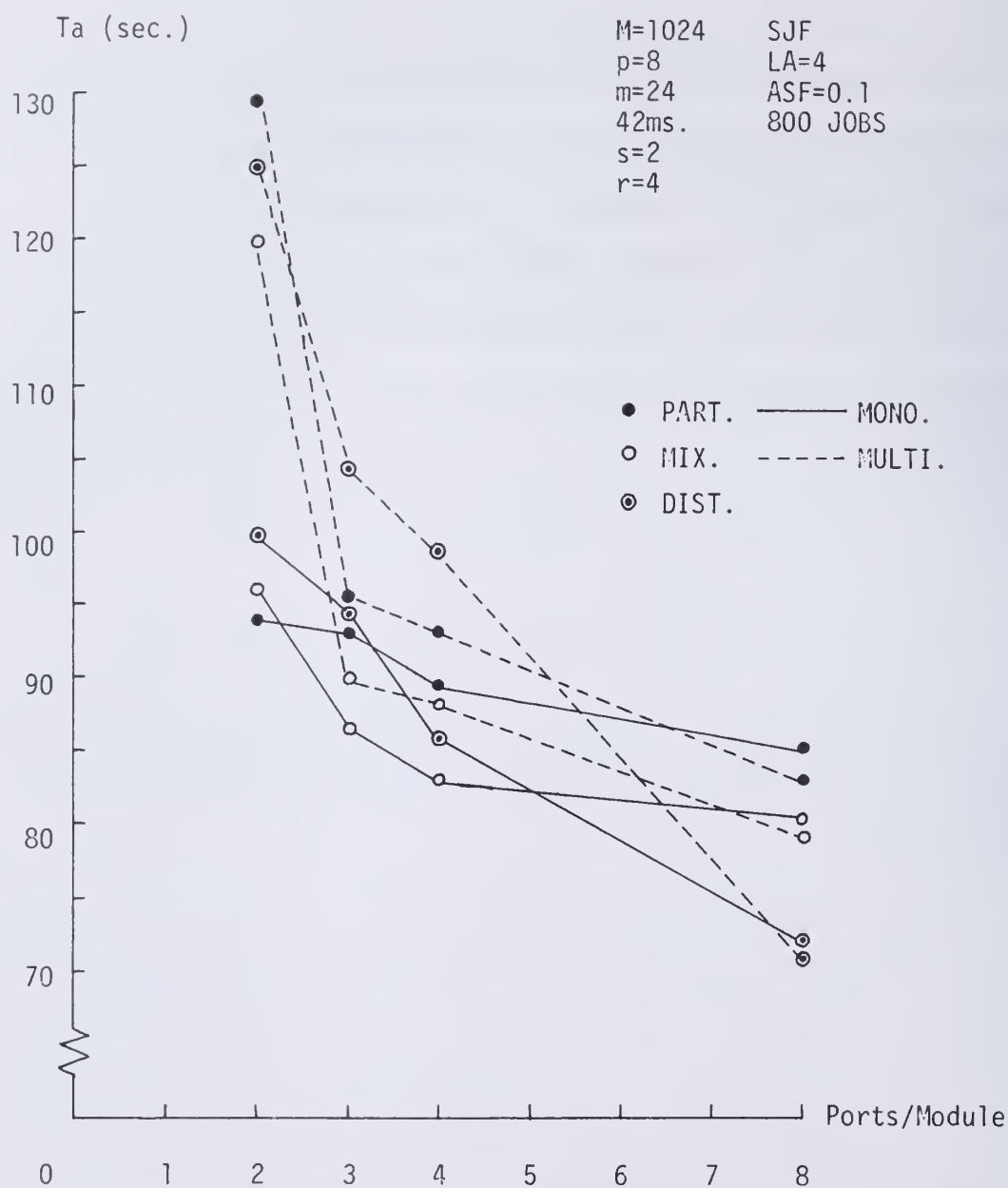


Figure 39. The Performance Degradation of the Partial Connection System.

Similarly, we use a (4,4,4,4,4,4,12,12) connection when the number of ports is 2. This network is connected in the same way as the last one except now the first six processors are occupying six disjoint groups of modules. Here is the connection matrix for this connection:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

All three of these partial connections are chosen simply because they are very symmetric. Of course, there are many other possible connections in each case, and we will look at some of them later.

In Figure 39, we show the turnaround time curves for all six combinations. As we can see, all six curves deteriorate when we reduce the number of ports of each module. However, many interesting and important results are shown in this figure which we are going to point out here.

Perhaps the most noticeable result is what happens to the two groups of curves, namely, the monoprogramming curves (the solid ones) and the multiprogramming curves (the dotted ones). When we use full connection (number of ports equal to 8), the multiprogramming results are all better than their corresponding monoprogramming results. But as we reduce the number of ports, the multiprogramming curves get worse rather quickly. For example, when we reduce the number of ports to 2, the distributed system curve increases by about 75% which is the worst among all six curves. The mixed system and partitioned system curves increase by 50% and 53%, respectively.

In the meantime, the monoprogramming curves increase by relatively small percentages, 39% for the distributed system curve, 21% for the mixed system curve, and only 10.6% for the partitioned system curve. When the number of ports is reduced to 2, all the monoprogramming results are well below the multiprogramming results. The monoprogramming curves win by a margin of roughly 30%. Obviously, multiprogramming is more sensitive to the connection.

The degradation of a monoprogramming curve can be explained by the memory waste caused by the reduction of connections. Table 19 shows the word memory utilizations of all the systems in Figure 39. The difference between the utilizations of a partially connected monoprogrammed system and a fully connected monoprogrammed system is, of course, the percentage of memory being wasted due to the reduction of connections. As we can see, in all the monoprogramming rows, the memory utilization is strictly decreasing, which means the memory waste is increasing. However, the memory waste is rather small. This is why the degradations of the monoprogrammed systems are small. Apparently, the reduction of connections only has little effect on the memory utilization of a monoprogrammed system.

Table 19, however, does not tell us the memory waste of a multiprogrammed system since all the memory utilizations for multiprogrammed systems are increasing. This does not mean that no memory is wasted in a multiprogrammed system. Some memory, although it might be small, must be wasted when we reduce the number of ports. Since a module can only be used by part of the processors, it is very likely this module will not be used up when the processors that attach to it are all occupied. Therefore, some other factor is causing the increase in the memory utilization in a partial

Number of Ports per Module System		8(Full)	4	3	2
Partitioned	Mono.	52.5	52.0	51.5	51.1
	Multi.	52.0	53.5	54.2	64.7
Mixed	Mono.	53.4	52.7	52.6	51.6
	Multi.	53.8	56.2	57.0	68.1
Distributed	Mono.	51.5	51.0	50.4	50.2
	Multi.	52.2	53.5	58.1	66.5

Table 19. The Word Memory Utilization of All the Systems in Figure 39.

connection multiprogrammed system.

In fact, it is not difficult to see. The memory utilization goes up because every job is now using the memory for a longer time. We have explained a similar phenomenon in Section 3.1.3 when we discussed Table 7. Let us show the average service time, i.e., the time a job spends in the memory of each multiprogrammed case in Table 20. As we can see, the service time t_s increases rather rapidly as we reduce the number of ports. The longer residence of each job in the memory thus results in a higher memory utilization. This is why the memory utilization is increasing instead of decreasing. Apparently, the memory waste due to the partial connection has been covered by this increase. Therefore, we cannot use memory waste to fully explain the degradation of the average turnaround time of a multiprogrammed system.

Interestingly, when we look at the statistics gathered from simulation outputs, we find that the queueing time a job spends in waiting for a processor increases almost at the same pace as the average service time increases. In other words, the increase of the service time comes from the increase of the queueing time waiting for a processor. In Table 20, we show this queueing time $\bar{q}^p$ together with the average service time.

From Table 20, we can see the queueing time for a processor can be as large as 23% of the total service time (10.1/82.2 for the distributed system). Obviously, this queueing increase is a big factor that causes the performance degradation of a multiprogrammed system. However, there is no queueing time for a processor in a monoprogrammed system since each job has its own dedicated processor. The degradation of a monoprogrammed system simply comes from the memory waste. This explains why the monoprogramming

System Parameter \ Number of Ports per Module		8(Full)	4	3	2
Partitioned	t_s	68.4	69.6	70.5	81.2
	$\bar{q}^p$	0.3	1.6	2.6	13.6
Mixed	t_s	70.6	73.3	74.7	87.7
	$\bar{q}^p$	0.4	3.2	4.8	18.1
Distributed	t_s	63.9	70.3	74.5	82.2
	$\bar{q}^p$	0.8	7.2	11.4	19.1

Table 20. The Average Service Time (t_s) and Queueing Time for Processors ($\bar{q}^p$) of the Multiprogrammed Systems.

curves deteriorate more slowly than their multiprogramming counterparts. The serious queueing delay when the number of ports is small, especially when it is 2, is the major reason why the multiprogramming curves are significantly higher than the monoprogramming curves.

Now, let us explain the cause of this queueing delay. Since in a multiprogrammed system there might be more jobs in the memory than the number of processors, the competition of a free processor is bound to happen. If all the processors are busy when a job comes in or returns from I/O, this job certainly will have to wait until some processor becomes free. The situation in a full connection system is simple since a job can be executed by any processor. So, all the pending jobs will wait in a single queue and get served on a first-come-first-serve basis. In a full connection case, the probability that all processors are executing jobs is rather small, since in the first place the probability that more than eight jobs in the memory is not too large, and secondly, a job will spend a significant amount of time in doing I/O. Therefore, the queueing time for a free processor will be small in this case. As we can see in the first column of Table 20, this is indeed the case. Moreover, since the partitioned system allows the smallest number of jobs in the memory, its queueing time is the smallest among the three allocation schemes.

However, the situation is much more complicated in a partial connection network. A job cannot be executed by every processor since not every processor can access all the memory modules this job occupies. In fact, the number of processors that can execute a job is bounded by the number of ports of a memory module. Very often, a job can only be executed by one particular processor! This is especially true when the number of

ports is small. Let us look at an example in Figure 40, where we show a portion of a partial connection, multiprogrammed system. As we can see, job c can be executed by both processors 3 and 5, but jobs a and b can only be executed by processors 5 and 4, respectively. So, for example, job "a" cannot be executed if processor 5 is executing another job. In other words, a job can only queue for the few processors that can execute it in a partial connection, multiprogrammed system. This certainly will cause a serious queueing delay. If we keep reducing the number of connections this situation will get worse and worse. This is why the queueing time for processor grows so fast when the port number is decreased, which in turn degrades the turnaround time.

Therefore, the monoprogramming scheme is more superior than the multiprogramming scheme if we are using partial connection, especially when the number of ports per module is small. This contradicts what happens in full connection systems.

Now, let us concentrate on the monoprogramming curves in Figure 39. In the full connection case, we can see the partitioned scheme yields the worst result and the distributed scheme yields the best. We have explained this in terms of memory bandwidth in the first part of this chapter. However, the situation starts changing when we use the partial connection. In the two-port memory connection, it is completely reversed, i.e., the partitioned scheme shows the best turnaround time, and the distributed scheme shows the worst. This phenomenon again can be explained in terms of memory bandwidth.

Let us reuse the example in Figure 40. In fact, what we show there is the picture of a distributed system if we assume the jobs are

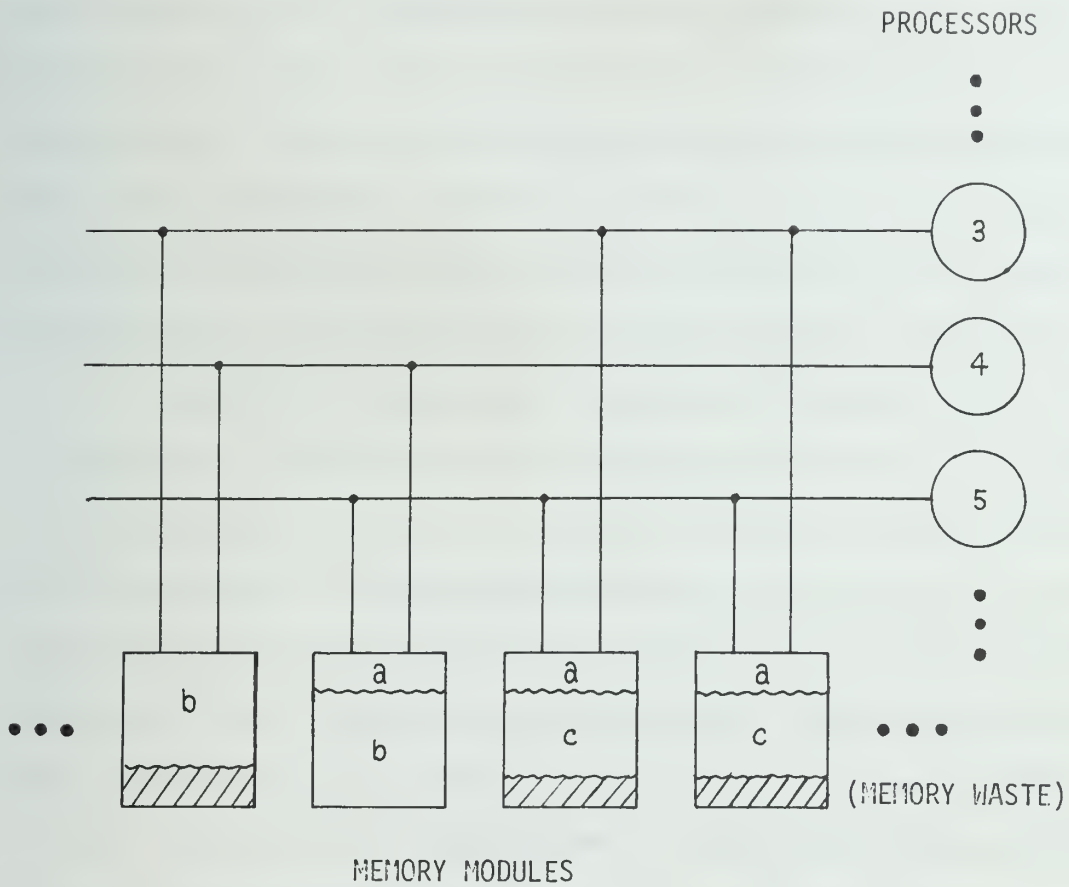


Figure 40. An Example of a Partial Connection, Multiprogrammed System.

allocated memory in the order a, b, and c. The biggest difference we can see is the degree of interleaving each job can have. For example, job a can only be interleaved in three modules if processor 5 does not connect to any other module. However, in a full connection, each job can be interleaved into as many modules as the system has. This reduction of the degree of interleaving drastically decreases the memory bandwidth of the distributed system. If, on the other hand, we use the partitioned scheme or the mixed scheme, each job can also be allocated and interleaved in a similar number of modules, although in general a little bit fewer. That means both the partitioned scheme and the mixed scheme can have bandwidth comparable to the distributed scheme. This is particularly true when the number of ports per module is small. But, the most important thing is, in a partitioned system there is no memory conflict between any two jobs. In the other two schemes, especially the distributed scheme, memory conflict will occur in those shared modules, which results in the degradation of the memory bandwidth. This is why the partitioned system shows the best turnaround time on the left half of the figure.

Therefore, we can come to the following two conclusions about the partial connection. First, monoprogramming is better than multiprogramming due to no queueing for processors in the former case. Second, partitioned scheme outperforms the other two schemes due to no memory conflict. Interestingly, both of these results completely reverse the situation in the full connection system. In the first part of this chapter, we kept emphasizing the advantage of monoprogramming and the partitioned scheme. However, the slightly better performance by multiprogramming and the distributed scheme tends to make these advantages look rather debatable. But, no doubt about

it, a partial connection system, monoprogramming and partitioned scheme are better from every aspect.

Perhaps the most important result in Figure 39 should be the small degradation of the partitioned, monoprogrammed system. When we reduce the number of ports from 8 to 2, the curve only increases by 10.6! Hence, we save 75% of the cost of the connection network but sacrifice only 10.6% of the performance. This is a tremendous improvement on the cost-effectiveness. Therefore, from a cost-effectiveness point of view, the partial connection system is a better architecture for system design.

Now, let us discuss, from the memory utilization point of view, why such a low degradation can be achieved. Recall in Figure 39, we used a (4,4,4,4,4,4,12,12) connection when each module has only two ports. Each of the first six processors connects to four modules, and no two processors connect to the same module. This essentially partitions the whole system into six disjoint subsystems as far as these six processors are concerned. We can see this from the connection matrix we showed earlier. Of course, these processors then can only handle jobs of size less than or equal to four modules.

Table 21 shows the job size distribution of the job mix we are using, where each job is counted toward the number of modules it will require under the partitioned scheme. We can see 81.3% of the jobs are of size less than or equal to four modules (170K bytes). So, most of the jobs can be handled by these six processors. Only the remaining 18.7% of the jobs, i.e., the large jobs, have to be handled by the other two processors, where each of these two processors is connected to 12 modules (512K bytes). We let each of these two "large" processors share memory with three of

Number of Modules	Density
1	.165
2	.146
3	.326
4	.176
5	.105
6	.010
7	.031
8	.019
9	.009
10	.003
≥ 11	.010

Module Size = $42 \frac{2}{3}$ K Bytes

Table 21. The Job Size Distribution.

the other six "small" processors. This arrangement certainly might cause some trouble for the large jobs. If a large job is under consideration for entering the memory but none of these large processors have enough room, even though together they have enough free space, we have to delay this job until a processor has gained enough memory by itself. In other words, a bad distribution of the small jobs in the memory might block a large job from entering. For example, if four 4-module jobs have occupied the first, second, fourth, and fifth processors, respectively, a 5-module still cannot enter since none of the last two processors can allocate a chunk of five modules to this job. So, a large job will experience more difficulties than it does in a full connection. This arrangement also has its own advantage. Apparently not all of the jobs running on the small processors will use exactly four modules, the unused space can be chained together by the large processor to make room for another job. Of course, most likely a small job will be chosen again since the space usually might not be large enough for a large job. So, a small job which requires four memory modules or less in fact can be assigned to any processor in the system.

Since the small jobs constitute an absolute majority of the job mix, the (4,4,4,4,4,4,12,12) connection should still allow a pretty good memory utilization due to the reason we mentioned above. Table 12 indeed shows that the memory utilization of this connection only degrades a little bit from the utilization of the full connection system. This is the reason why the turnaround time increases by just a small percentage.

From Table 21, we can see that 10.5% of the jobs require more than four modules but no more than five modules. People might wonder

if the system should perform better if we assign a few more modules to some of the small processors so they can handle this 10.5% of the jobs and alleviate the traffic in front of the two larger processors. We have collected the results for several slightly different connections, and we found the results are more or less the same as that of the (4,4,4,4,4,4,12,12) connection, despite the inclusion of some 5-module processors. For example, Table 22 shows the results for two other connections, (3,3,4,4,5,5,12,12) and (3,3,3,5,5,5,11,13), which are very similar to the first connection. Apparently, the memory sharing of a large processor with three 4-module processors can take care of the 5-module jobs very well. But most importantly, we gain some confidence that these results are indeed in a reasonable region.

Of course, the job size distribution is a very important factor to the performance of a partial connection. As we said earlier, 81.3% of the jobs are of size less than or equal to four memory modules which is the major reason why the (4,4,4,4,4,4,12,12) connection can have good performance. However, if we increase the size of each job, the performance of this connection might degrade rather severely since more jobs now have to enqueue for those two large processors. In Table 23, we show some results of monoprogrammed, partitioned system when we increase the job size by 25% and 50%. We can see the turnaround time of the (4,4,4,4,4,12,12) connection indeed degrades very quickly. It increases by 33% when we increase the job size by 25%. In the last row of that table, we can see that the percentage of the jobs with sizes less than or equal to four modules is now reduced down to 75.9%, which obviously is the reason of the performance degradation. If we increase the job size by 50%, we can see the turnaround

Job Size Scaling Factor Connection	1.00	1.25	1.50
(4,4,4,4,4,4,12,12)	.94	126	237
(3,3,4,4,5,5,12,12)	95	119	162
(2,2,5,5,5,5,12,12)*	106	128	137
% of Jobs with Sizes ≤ 4 Memory Modules	81.3	75.9	58.6

* Connection Matrix :

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Table 23. The Effect of Job Size on the Average Turnaround Times of Three Different Connections.
(Monoprogramming, Partitioned Scheme)

time increases drastically to 237 which is 152% higher. This is because 41.4% of the job now requires more than four modules of memory. Apparently, the system is now saturated under this circumstance.

If we use (3,3,4,4,5,5,12,12) connection, i.e., assign one more module to each of the fifth and sixth processors, we can see the situation is much better when we increase the job size. It only degrades by 70% if we increase the job size by one-half. In fact, we can see in Table 23, the (2,2,5,5,5,5,12,12) connection gives us the best result for enlarged job size. It degrades just 30% for a 50% increase of the job size. In other words, using more 5-module processors can result in a smaller degradation.

Therefore, how to assign memory modules to processors really depends on how the job size distributes. It is very difficult to formulate an equation and try to solve an "optimal" solution of a partial connection. The only rule of thumb is to look at the job size distribution and partition the memory ports so that enough processors can have sufficient memory space to handle most of the jobs. In other words, try to assign the memory so that no severe bottleneck will be created at any processor. For example, before we scale the job size, four memory modules will be sufficient for a processor since 81.3% of the jobs are smaller than or equal to four memory modules. However, when we increase the job size, we need to connect more modules to some processors so they can handle larger jobs. Our results indeed show that this approach is generally correct. Of course, some of the processors will obtain less memory because the total number of ports is fixed.

For more than two ports per memory, the same kind of approach

can also be used, except each processor can then connect to more memory modules. The memory utilization will be better, and hence the performance will be improved.

One more interesting thing about the result shown in Table 23 is that the performance of a partial connection system is very sensitive to the increase in job size. For example, for a 50% increase, the average turnaround times of these three connections increase by 152%, 70%, and 30%, respectively. If we refer back to Table 12, we can see the turnaround time of a full connection system only degrades 15% when we increase the job size by 50%, which is significantly lower. So, when we are planning to use a partial connection network, we ought to be very careful about the job size distribution and use enough memory in order to achieve a satisfactory level of performance.

Finally, let us redo Figure 35 for the partial connection system, i.e., find out the effect of system size on the system performance. Figure 41 shows how the average turnaround time changes when we double the system size. The solid curves are using an arrival rate ratio of 2, just as we did in Figure 35. We again use the (4,4,4,4,4,4,12,12) connection and an arrival scaling factor of 0.1 for the (8,24,4) system. For the (4,12,2) system, we use a (4,4,4,12) connection, which is exactly one-half of the (4,4,4,4,4,4,12,12) connection, and an arrival scaling factor of 0.2. For the (16,48,8) system, we use a (4,4,4,4,4,4,4,4,4,4,4,4,12,12,12,12) connection and an arrival scaling factor of 0.05. So, we again double the workload when we double the system size. As we can see, the T_a curves drop quickly when we double the system size, which is very similar to that in Figure 35. But, the interesting thing is the curve of the

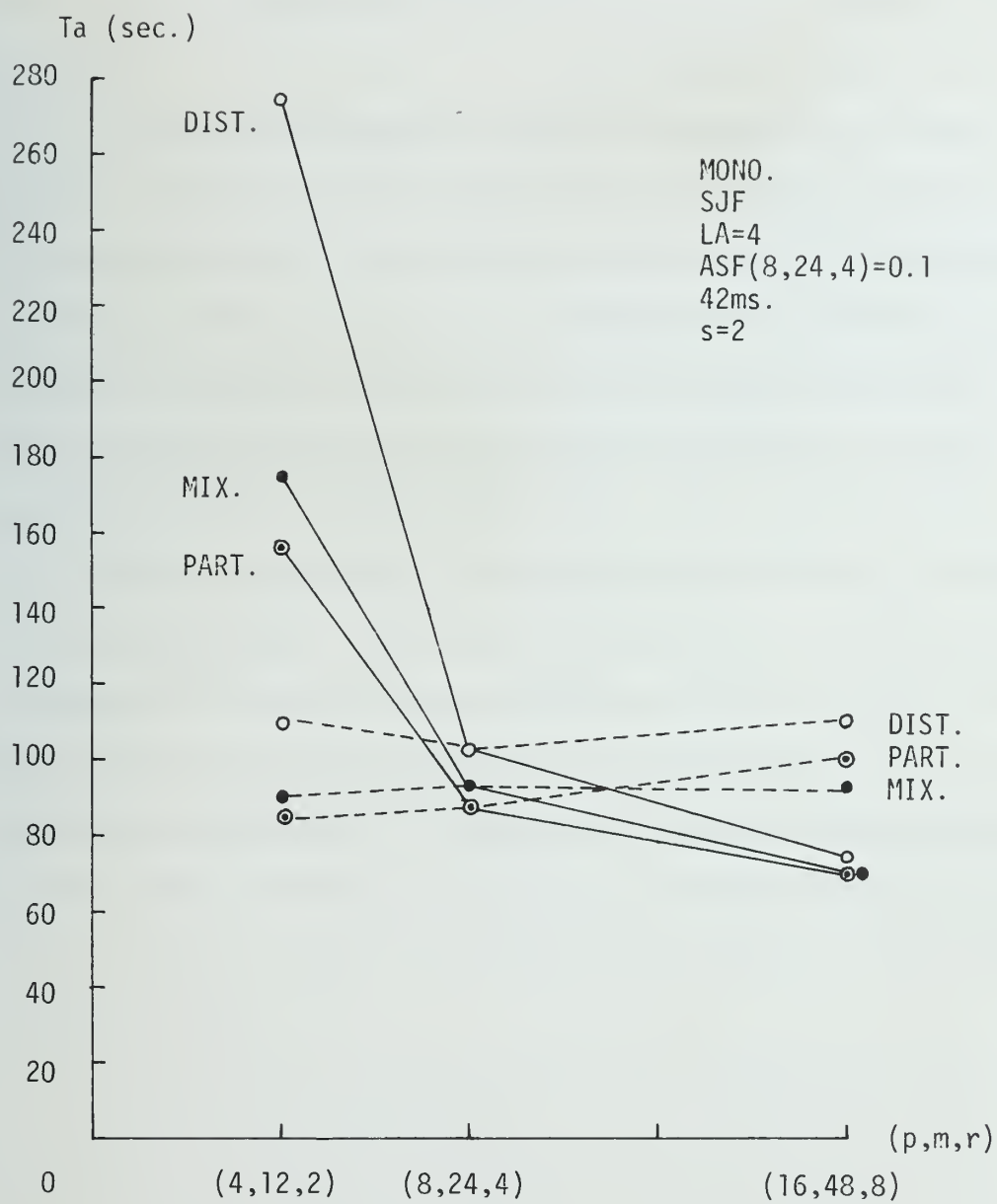


Figure 41. The Average Turnaround Time for Different System Size. (Use Partial Connection)

distributed system now is most sensitive to the reduction of system size. In Figure 35, however, the curve of the partitioned system degrades quickest when we decrease the system size.

Again, we increase the arrival scaling factor of the (4,12,2) system in order to lower its turnaround time. Very surprisingly, when we use 0.28, which is the same scaling factor we used in Figure 35, the turnaround time of the (4,12,2) system drops close to that of the (8,24,4) system. On the other hand, when we decrease the arrival scaling factor of the (16,48,8) to 0.0395, the turnaround time becomes roughly the same as that of the (8,24,4) system. This implies that our $2^{1.5}$ conjecture still holds in a partial connection system. Of course, the result we get here is based on one particular set of connections. Although the performance of a partial connection system is very sensitive to the connection we use, at least we know it is possible to connect the system so that a double-sized system can carry a workload $2^{1.5}$ times of the workload of the original system. Hence, we now gain more confidence in this simple conjecture.

Chapter 4

CONCLUSION

4.1 Summary

In the last chapter, we have discussed several interesting problems about the design of a multiprocessor system. We talked about the performance of multiprogramming and monoprogramming schemes, the advantages and disadvantages of three different memory allocation schemes, the effects of job parameters and hardware characteristics, and the difference between using a full connection network and a partial connection network. We will briefly summarize these results in this section.

Tables 24-27 summarize and compare the performances of six system combinations under both full and partial connections. Each table will show the comparison of one performance parameter. It is rather difficult to order the performance of various systems, so we will only use {bad, fair, good, best} or {high, moderate, low} to indicate their relative performances. However, we do point out the system which yields the best results in order to give the reader an idea which system might be the best choice for each area of performance.

Table 24 shows the comparison of the average turnaround time. Of course, the turnaround time of a monoprogrammed system depends heavily on the number of processors. We will assume that there are enough processors, say 8, in the system. Under a full connection, the distributed, multiprogrammed system has the best turnaround time. As we explained earlier, this is caused by high memory bandwidth and high memory utilization. The distributed, monoprogrammed system has the next best turnaround time. Obviously, this is because we are assuming eight processors in the system. If we assume fewer

System Connection		Full	Partial ^{**}
Mono-programming	Partitioned	Bad	Good
	Mixed	Good	Good
	Distributed	Near Best [*]	Good
Multi-programming	Partitioned	Fair	Fair
	Mixed	Good	Fair
	Distributed	Best	Bad

* Assuming 8 Processors

** Assuming 2-Port Memory

Table 24. Comparison of the Average Turnaround Time.

processors, the turnaround time will degrade a little until we reduce the number of processors below four (cf. Figure 32). So, both distributed systems perform very well if we use a full connection network. Both mixed systems also have good turnaround times but are worse than the distributed systems. This is due to a smaller memory bandwidth produced by the mixed scheme. On the other hand, the partitioned systems both perform even worse than the mixed systems, although not by much. This is caused by bad memory utilization and memory waste of the partitioned system. Overall, a multiprogrammed system is slightly better than its monoprogrammed counterpart, and the distributed scheme yields the best result. The performance of a full connection system is essentially determined by the memory utilization and the memory bandwidth.

Under a partial connection, however, the whole situation is reversed. All the monoprogramming results are better than the multiprogramming results when the number of ports is reduced to two. As we explained in the last chapter, the major reason is the queueing time for processors created in the partial connection, multiprogrammed system. But in a partial connection, monoprogrammed system, there is no queueing time for processors since each job is assigned a dedicated processor. We can see that the partitioned scheme yields the best result in a partial connection, monoprogrammed system. The interesting thing is that it has the worst performance in a full connection, monoprogrammed system. So, the connection has really changed the result. The mixed, monoprogrammed system also shows very good performance which is similar to the partitioned, monoprogrammed system. In fact, all monoprogrammed results are very close to each other. We can see this from the results shown in Section 3.2.4. On the other hand, all the multiprogrammed

systems perform relatively poorly when we use a partial connection.

The most important result is, if we properly interconnect the processors and memories, the performance degradation of a partial connection, monoprogrammed system can be kept to within 10 to 20% of the performance of a full connection system. For example, we have shown that the (4,4,4,4,4,4,12,12) connection only creates 10.6% of degradation on the turnaround time of the partitioned, monoprogrammed system. This not only encourages us to use partial connection since it is more cost-effective, but also makes the partitioned scheme and monoprogramming more attractive in operating system design.

Table 25 shows the comparison of total memory bandwidth, i.e., the memory bandwidth generated by all active processors. Under a full connection, the distributed scheme can yield relatively high memory bandwidth due to the high degree of interleaving each job can enjoy. The memory bandwidths of both the partitioned and mixed schemes are lower than that of the distributed scheme since now each job is confined in part of the memory and the degree of interleaving has been reduced. As we said, this is the major reason why the distributed system can have better turnaround time than the other two systems. However, if we use faster memory, i.e., reduce the value of s , the difference between the bandwidths of the distributed and partitioned systems will be reduced. Their turnaround times hence become closer to each other. This can be seen in Figure 36. Of course, the total memory bandwidth of a multiprogrammed system is higher than that of its monoprogrammed counterpart since the multiprogrammed system on the average can contain more active jobs in the memory.

Under a partial connection, the total memory bandwidth of every

System Connection		Full	Partial
Mono-programming	Partitioned	Low	Low
	Mixed	Moderate	Moderate
	Distributed	High	Moderate
Multi-programming	Partitioned	Moderate [*]	Low
	Mixed	High	Moderate
	Distributed	Highest	Moderate

* Assuming Large m
(Low for Small m)

Table 25. Comparison of the Total Memory Bandwidth.

system will decrease. This is due to the decrease of memory utilization. For a distributed system, the memory bandwidth has been further decreased by the reduction of the degree of interleaving since a job can no longer be interleaved across the whole memory in a partial connection. Now, the total memory bandwidths of the mixed and distributed systems are similar because they have similar capability of containing jobs and similar degree of interleaving for each job.

One thing we need to explain is the total memory bandwidth of a partial connection system. Intuitively, a multiprogrammed system should have a higher total memory bandwidth than its monoprogrammed counterpart. This is because a multiprogrammed system can allow more jobs in the memory at the same time which can cause a higher utilization of processors. However, our simulation result shows that a monoprogrammed system has almost the same total memory bandwidth as a multiprogrammed system if we use partial connection. This rather surprising result actually is not difficult to explain. As we said in Section 3.2.3, the major factor that makes a partial connection, multiprogrammed system have a worse turnaround time than its monoprogrammed counterpart is the queueing time for processors (see Table 20). This queueing time is caused by the fact that in a partial connection system every job can only be executed by a few processors which connect to all the memory modules this job is in. In other words, a job will have to wait if the processors that can handle it are all busy, even though some other processors are free. This queueing phenomenon essentially reduces the number of jobs that can be executed at the same time. Our simulation result shows that while there are more jobs in a multiprogrammed system, on the average both systems have approximately the same number of jobs in execution. The

extra jobs in the multiprogrammed system are either in the processor queue or in the I/O stage. Since the numbers of jobs in execution are roughly the same, both systems thus have similar total memory bandwidth.

However, we know the total memory bandwidth represents the amount of work a system can do in one memory cycle time. It actually indicates how good the system throughput is, if we consider the system throughput as the number of jobs that get done in a certain unit of time. The higher the total memory bandwidth is, the faster the jobs can be done, and hence the higher the system throughput will be. So, the partial connection, multiprogrammed system should have the same throughput as its monoprogrammed counterpart does. Our simulation result indeed shows this since both of them have very similar total elapsed time, i.e., the total amount of time to finish all the jobs. In other words, the use of multiprogramming does not improve the system throughput if we are using a partial connection.

As we said, the partial connection, multiprogrammed system has a worse average turnaround time due to the occurrence of the queueing time for available processors. Yet, it has the same throughput as the partial connection, monoprogrammed system, which means it can finish the jobs at the same rate. If we look at the input and output of both systems, we can see both systems have the same arrival and departure rates. The only fact that can cause the difference of the average turnaround times is apparently the order these jobs get done. Although we are using the same scheduling algorithm in both systems to select jobs for execution, the queueing time for available processors in a partial connection, multiprogrammed system might delay the execution of some jobs and allow some other jobs to be processed faster. For example, a job that comes into the memory first and requires smaller

CPU time than any other job in the memory might not be finished first if it has to compete for a processor with some other jobs all the time. In a monoprogrammed system, this will not happen since the time a job will finish once it enters the memory solely depends on its CPU and I/O time requirements. In other words, the effect of the scheduling algorithm will be reduced by the queueing delay in a multiprogrammed system. This is why a partial connection, monoprogrammed system has a better average turnaround time.

Therefore, the average turnaround cannot always tell us how fast the system is doing work. Only the total memory bandwidth can indicate how good the system throughput is.

Let us now summarize the memory utilization of these systems in Table 26. As we can see, the multiprogrammed systems have better memory utilization than the monoprogrammed systems, and the full connection systems have better memory utilization than the partial connection systems. This is what we would expect. From all the data we collected, the mixed and distributed systems both show rather similar memory utilizations. This is because both systems have the same capability of containing jobs in the memory, as we have mentioned several times. The partitioned system, however, has a significantly lower memory utilization, which is caused by the memory waste created during the whole-module allocation process. This is the major reason why the partitioned system yields the worst turnaround time when we are using a full connection network.

Of course, a partial connection system has lower memory utilization than a full connection system since only some of the processors are connected to a memory module. So, if all the processors connected to a certain module are busy, then the unused portion of this module will be wasted. Or, if the

Connection System		Full	Partial
Mono-programming	Partitioned	Bad	Bad
	Mixed	Good	Fair
	Distributed	Good	Fair
Multi-programming	Partitioned	Good	Fair
	Mixed	Best	Good
	Distributed	Best	Good

Table 26. Comparison of the Memory Utilization.

unoccupied memory is split between several processors and none of these processors has by itself large enough space for the next job, then the unoccupied memory again will be wasted. This is what causes the performance to degrade. Fortunately, if we can use a good connection by considering the job size distribution, it is possible to keep the memory waste, and hence performance degradation, very low.

The other performance parameter we often mentioned in the last chapter is the job memory bandwidth, which is the bandwidth each processor gets to execute a job. It is obtained by dividing the total memory bandwidth by the number of jobs in memory. As it turns out, the job memory bandwidth of the mixed and partitioned systems are only affected by the processor-memory speed ratio and the number of memory modules m . They remain essentially unchanged when we change the other system parameters. This is not surprising, since under these two schemes most or all of a job is isolated and prevented from the influences of the other jobs. Once a job gets into the memory, the speed ratio will affect the memory bandwidth this job can get since s determines the number of requests the processor can generate per memory cycle. On the other hand, m will determine the degree of interleaving for a job which also affects the bandwidth. However, the job memory bandwidth of the distributed system will be affected by almost every system parameter. Of course, it will be affected by the speed ratio s and the number of memory modules m . In addition, it will also be affected by parameters like job arrival rate, the average amount of time per I/O operation, the average number of I/O requests per job, monoprogramming or multiprogramming, and so on. All these parameters will affect the number of jobs in execution at the same time, which in turn will affect the memory bandwidth

due to mutual interference. Furthermore, the connection network also has a very significant effect on the job memory bandwidth. As we said before, a partial connection might drastically reduce the degree of interleaving of a job and will seriously decrease the job memory bandwidth.

Overall, a full connection distributed, monoprogrammed system will yield the highest job memory bandwidth. This can be seen in Table 7 where we show some numerical values of the job memory bandwidth.

In Table 27, we list the system which produces the best result in each performance area, under either a full or a partial connection. If we use full connection, the distributed, multiprogrammed system shows the best turnaround time, the largest total memory bandwidth, and the highest memory utilization. Only the distributed, monoprogrammed system displays the best job memory bandwidth. On the other hand, if we use partial connection, the partitioned, monoprogrammed system now shows the best turnaround time. Both the distributed, multiprogrammed system and the distributed, monoprogrammed system display the best total memory bandwidth. The best memory utilization and the best job memory bandwidth are still obtained by using the distributed (or mixed), multiprogrammed system and the distributed, monoprogrammed system respectively.

The full connection, distributed, multiprogrammed system seems to be a better choice, since it gives the minimum turnaround time. However, the partial connection, partitioned, monoprogrammed system is more cost-effective. Especially when the system size is large, the use of a partial connection network can reduce the system cost significantly. Moreover, a partially connected system is easier to maintain and expand. While we are adding or deleting a memory module or a processor, only very few connections have to

System Performance	Full Connection	Partial Connection
Turnaround Time	Distributed, Multiprogrammed	Partitioned, Monoprogrammed
Total Memory Bandwidth	Distributed, Multiprogrammed	Distributed, Both
Memory Utilization	Distributed or Mixed, Multiprogrammed	Distributed or Mixed Multiprogrammed
Job Memory Bandwidth	Distributed, Monoprogrammed	Distributed, Monoprogrammed

Table 27. Systems with Best Performance.

be altered, and the rest of the system can be kept untouched and go on operating. So, a partial connection system also has the advantage of high availability and expandability.

All the performance measures, in particular the turnaround time, are very sensitive to the job mix we are using. We have shown the turnaround time and the memory utilization will increase rather rapidly when we increase the arrival rate, job size distribution, or the I/O time. The reason is, these parameters can easily push the system into saturation. Therefore, when we are designing a system, we should carefully study the job mix we are dealing with.

One of our most interesting results is the $2^{1.5}$ workload relationship between two systems that have a size ratio of 2. Our simulation shows that, when we double the system size, we can handle 2.7 to 2.8, or roughly $2^{1.5}$, times the original workload. This is true for both the full and partial connection systems. So, our conjecture is, the system size C (or the cost) and the workload it can handle P (or the processing power) maintain a $P = a C^{1.5}$ relationship. Of course, this conjecture has been shown to hold only for systems of size up to 16 processors and 48 memory modules. As we said in Section 3.2.3, this factor will be reduced to about 2.3 when we double the system size again to (32, 96, 16). So, we believe the improvement factor of 2.8 would approach 2 as the system gets very large.

4.2 Some Design Problems

4.2.1 Address Interleaving

As we said in the last section, a partial connection, partitioned, monoprogrammed system is the most cost-effective choice of system design.

However, we pointed out that we will need a new scheme of generating physical addresses if we want to use interleaving to get the best possible memory bandwidth. Of course, when the memory-processor speed ratio s is small, say 1 or 2, there will not be much difference whether we use interleaving or not. For $s=2$, even if we store a program vertically inside a memory module, quite often we might still be able to access more than one word if the data and the instruction we can fetch simultaneously are in different modules. If we use interleaving, i.e., store a program horizontally across several memory modules, we might only get a little better chance of accessing two words without conflict. So, it might not be worth it to implement the interleaving scheme when s is small. When s is larger, however, the interleaving scheme will show a much better bandwidth since several instructions and data can be accessed at the same time. It is more desirable to use interleaving under this circumstance.

If we use interleaving in a partitioned system, two problems will arise that make the generation of physical addresses very tough. First, the number of memory modules allocated to a job is variable depending on the size of this job. This implies that the degree of interleaving will be different for each job. Consequently, a processor must be able to adjust its address mapping mechanism to cope with the changing degree of interleaving. Second, the modules a job gets in general will be scattered all over the memory and might not be adjacent to each other. Therefore, it will cause some trouble to locate an instruction or operand. If we horizontally interleave a program, the next instruction might be several modules away from the current instruction. So, we will not be able to get the address of the next instruction by simply adding one to the current module

number. Apparently, we need more hardware and a new algorithm in the instruction decoding unit of a processor in order to generate a physical address properly. Let us propose a simple and feasible design which can solve this problem.

Figure 42 shows the logic diagram of the design we are proposing. The logical address register contains the logical address we want to transform and the final physical address will be in the physical address register. The hardware between them is used to do the transformation. The physical address consists of two parts, namely, a module number x and an in-module word address w , which will be obtained by the following process.

First, let us point out one thing which will affect the way we interleave a program, and hence affect the memory bandwidth. Assume that the program counter is L bits long. If we interleave a program successively into all the memory modules, say c of them, we must be able to perform a "quotient-remainder of c " operation on these L bits, called $QR_c(L)$, in order to find the module and word corresponding to this address. However, c is a variable which is determined by the size of the job currently running on this processor. This means every processor should be provided the hardware to perform the QR_c operations for all possible c values if we want to interleave a program in a normal way. This is not economically attractive since it implies that we must build several QR circuits inside each processor for address decoding. Therefore, we must seek some other method to interleave a program.

The method we suggest is the following: If a job requires a power of two modules or some number of modules for which QR hardware exists, then we will interleave the program in the normal way. Otherwise, we will

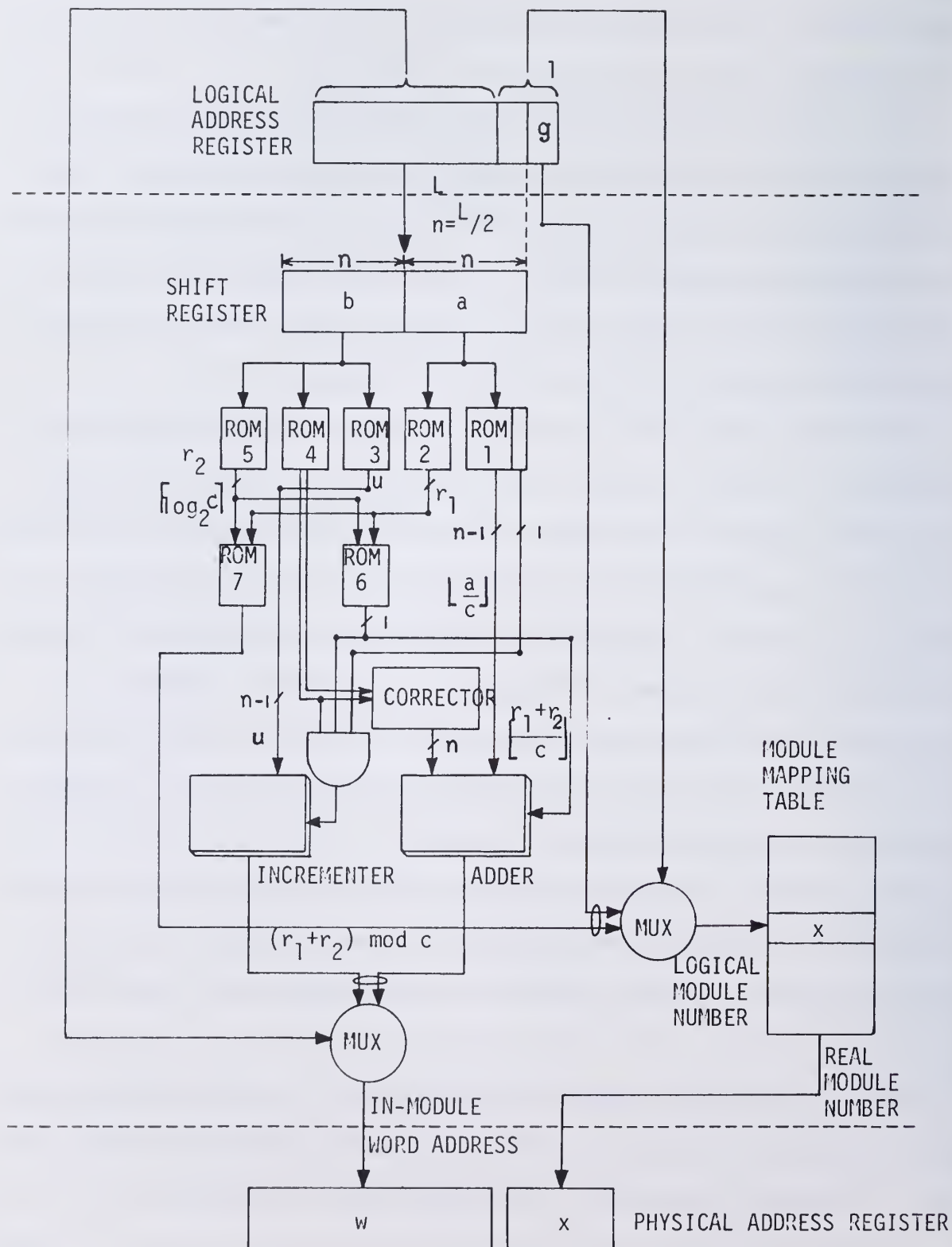


Figure 42. Address Mapping for Partitioned System.

partition the modules into a power of two groups each having the same number of modules such that QR hardware exists for this group size. For example, assume the processor has the hardware to do a QR_3 operation and a job requires six modules. We will partition the modules into two groups with three modules in each group. However, if the number of modules a job needs is not a multiple of 3, we will have to grant the job some extra memory to make the number a multiple. Now, we will use the last g bits of the logical address to determine the proper group. These g bits are called "Group Bits." If there are only two groups, g will be 1. In so doing, we actually achieve a double interleaving, i.e., we not only interleave the successive addresses into different modules, but also interleave them into different groups. Figure 43 shows the result of interleaving a 6-module job, if we use the last bit to indicate the group. We will call this a 3-3 interleaving. The important thing is that this method still allows us to fully interleave a program across all the modules even though we do not have the appropriate QR_6 circuit.

The first (left) $L-g$ bits of the program counter will be fed into a shift register where we will perform the QR operation. Since g is a variable which depends on the number of groups we form for the job, we need a shift register here in order to shift the logical Address g bits to the right. Of course, if g is zero the whole content of the logical Address register will be gated into the shift register without shifting. So, the shift register must also be L bit long. We now perform the QR operation on the content of the shift register. The remainder of this operation will tell us the correct module within a group. This remainder together with the group bits will tell us the logical module number we are looking for.

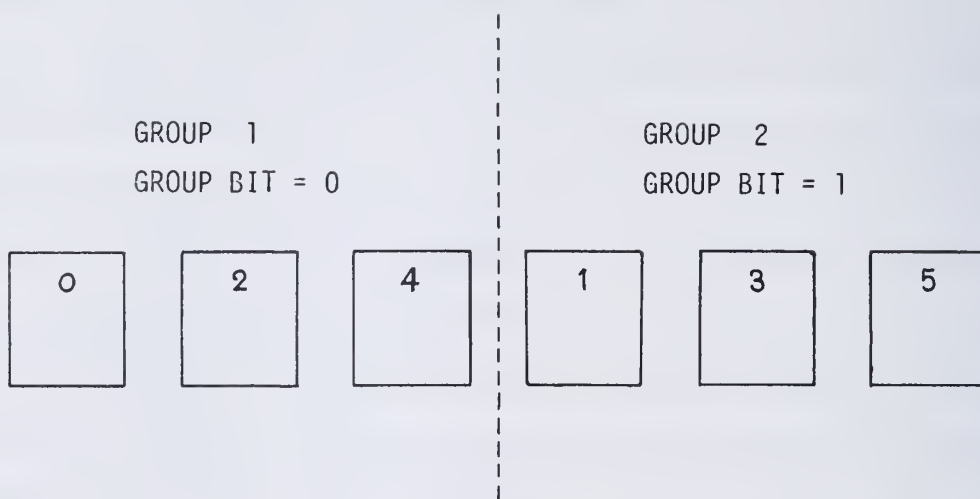


Figure 43. The Interleaving of a 6-Module Job.

The quotient will give us the correct address inside the module.

Now, let us describe how we do the QR_3 operation. Of course, we can use a combinational circuit to perform the operation. For example, Gajski and Vora [49] have a very nice design of modulo 3 circuit. However, it might take a large number of gates to implement the circuit when L is large, and we also need to determine the quotient. So, we choose another design using read-only-memories (ROM). If L is small, say 10 to 12, we can use one ROM to find out the remainder and quotient of 3. However, we are using 1024K bytes of memory in our system, which means L is about 18 to 20. We then have to use a very large ROM with bits in the order of 2^{20} ! This is apparently very expensive. So, we will use the design shown in Figure 42, which requires seven ROM's of reasonable size, one small integer adder and an incrementer. Of course, we have to spend a little longer time to do the operation.

In fact, our design is good for any base. Let us use c to represent the base of the QR operation. We first break the shift register into two equal halves each having $n = L/2$ bits. The contents of the right and left halves will be called a and b respectively. In other words, the content of the shift register can be expressed as $a + b2^n$. Also, let us further decompose b to $uc+v$, i.e., let $b = uc+v$. The remainder and quotient of the QR_c operation can be found to be:

$$\begin{aligned} \text{remainder} &= (a+b2^n) \bmod c \\ &= [a \bmod c + (b2^n) \bmod c] \bmod c \\ &= [r_1 + r_2] \bmod c, \end{aligned}$$

and

$$\begin{aligned}
 \text{quotient} &= (a + b2^n) \div c \\
 &= \left\lfloor \frac{a}{c} \right\rfloor + \left\lfloor \frac{b2^n}{c} \right\rfloor + \left\lfloor \frac{r_1 + r_2}{c} \right\rfloor \\
 &= \left\lfloor \frac{a}{c} \right\rfloor + u2^n + \left\lfloor \frac{v2^n}{c} \right\rfloor + \left\lfloor \frac{r_1 + r_2}{c} \right\rfloor,
 \end{aligned}$$

where $r_1 = a \bmod c$, $r_2 = (b2^n) \bmod c$, $\div$ represents integer division, and $\lfloor \cdot \rfloor$ is the floor function. ROM 2 contains the result of $r_1 = (a \bmod c)$ and ROM 5 contains the result of $r_2 = \lfloor (b2^n) \bmod c \rfloor$. The outputs of these two ROM's will be used as the address to ROM 7, which contains the result of $(r_1 + r_2) \bmod c$. The output of ROM 7, i.e., the remainder, coupled with the group bits will tell us the logical module number. Each word of ROM 1 contains a result of $\left\lfloor \frac{a}{c} \right\rfloor$ together with a bit that tells whether a is all one's or not. Hence, all the words in ROM 1 have zero in the last bit position except the last word. The use of this bit will be explained very soon. ROM 3 contains the result of u which will be shifted to the left by n bits. The third term on the right hand side of the quotient equation, i.e., $\left\lfloor \frac{v2^n}{c} \right\rfloor$, will be called a "Corrector" term. Since v has only c possible values, the number of possible values for the corrector term is at most c . If c is small, the corrector term can only have a few possible values. Therefore, we can either hardwire them or use a small number of registers to store them. ROM 4 will be used to select the corrector value we should use. The outputs of ROM 2 and ROM 5, i.e., r_1 and r_2 , will also be fed into ROM 6, which gives the result of $\left\lfloor \frac{r_1 + r_2}{c} \right\rfloor$. It is easy to see that $0 \leq r_1 + r_2 < 2c$, so $\left\lfloor \frac{r_1 + r_2}{c} \right\rfloor$ is either 0 or 1. Hence, the output of ROM 6 is only 1 bit long which can be used as the carry-in to the adder.

The adder is only n bits long which adds $\left\lfloor \frac{a}{c} \right\rfloor$, $\left\lfloor \frac{v2^n}{c} \right\rfloor$, and $\left\lfloor \frac{r_1 + r_2}{c} \right\rfloor$

together. These three terms are $n-1$, n , and 1 bits long respectively. The fourth term, i.e., $u2^n$, will be fed into an incrementer, which will increment by one if the adder generates a carry. The reason we use an n -bit adder and an n -bit incrementer instead of a $2n$ -bit adder is because the summing of these four terms is rather special. Figure 44 shows the length of each term and how they align when they are summed together. As we can see, the only chance u will be affected is when the other three terms produce a carry. So, we only need an incrementer for the left n bits. In fact, the carry can be generated in advance by using the last bit of ROM 1,

$\lfloor \frac{r_1+r_2}{c} \rfloor$, and the output of ROM 4, so we do not have to wait for propagation delay of the adder. This can be shown by the following example.

Let us assume $c=3$ and $n=10$. $\lfloor \frac{a}{c} \rfloor$ will be 9 bits long. Since v can only be 0, 1, or 2, the output of ROM 4 is only 2 bits long. The corrector term can be shown to be one of the following three values: namely, 0000000000, 0101010101, or 1010101010 (all in base 2). The last word of ROM 1 contains 101010101 which is the largest value of $\lfloor \frac{a}{c} \rfloor$ and is the only word that will generate a carry provided the corrector term is 1010101010 and $\lfloor \frac{r_1+r_2}{c} \rfloor = 1$. The last bit of ROM 1 will tell whether $\lfloor \frac{a}{c} \rfloor$ is 101010101 or not and the output of ROM 4 can tell whether the corrector term is 1010101010 or not. So, we can AND them together to see if we should increment u or not. The results of the adder and incrementer will be combined to form the quotient we are looking for.

The size of ROM 1 is $2^n \times n$ bits, the sizes of ROM 2, ROM 4, ROM 5, and ROM 7 are all $2^{\lceil \log_2 c \rceil} \times \lceil \log_2 c \rceil$ bits, the size of ROM 3 is $2^n \times (n-1)$ bits, and ROM 6 is only $2^n \times 1$ bit. In fact, ROM 6 and ROM 7 can

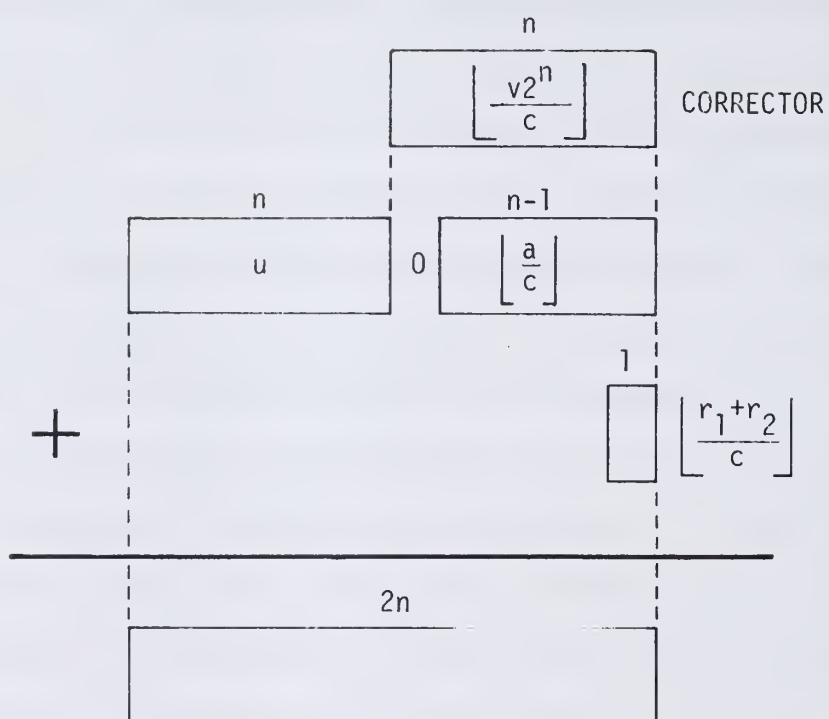


Figure 44. The Length of Each Component in Quotient Computation.

be put together in one ROM. If L is 20, n is 10, and c is 3, we need 20 $1K \times 1$ bit ROM's and one 16×4 bit ROM. So, the hardware is actually very cheap.

We can see from Figure 42 that it takes at most two ROM cycles and one addition cycle to do a QR operation. However, this is in general less than most of the arithmetic operation times. So, our design will not cause any serious problem to the address decoding.

As we mentioned earlier, the memory modules a job gets might be scattered all over the memory, and they might not be adjacent to each other. So, we need to transform the logical module number we obtained above to the physical module number. This is what the Module Mapping Table shown in Figure 42 will do. The Module Mapping Table is in fact a cache memory. When we allocate memory to a job, we will record the physical module numbers in the table in the correct order, which can be retrieved later. Hence, the Module Mapping Table acts just like the page table used in a paging system. After the physical module number has been retrieved, it will be appended to the in-module word address to form the final physical address.

Of course, a job that requires some power of two modules does not need the QR operation since the module number and the in-module word address can be obtained simply by breaking the logical address into two parts. the lower ℓ bits (cf. Figure 42) will indicate the module number which will be used by the Module Mapping Table to retrieve the physical module number. The upper $L - \ell$ bits will be gated directly to the physical address register. The two multiplexors (MUX's) in Figure 42 are used to decide which result should be used.

The only drawback with this scheme is sometimes we have to waste some memory in order to make this scheme work. For example, if we can only

do QR_3 operation and a job requires five modules, we must allocate six modules to this job and use the interleaving scheme shown in Figure 43. Table 28 shows the actual number of modules each job will be allocated and the interleaving scheme to be used. As we can see, 5-module and 7-module jobs will be granted six modules and eight modules respectively. Obviously, this design will further increase the memory waste originally present in a partitioned system. The situation is even worse for 9-module, 10-module, and 11-module jobs since all of them must be allocated 12 modules to use 3-3-3-3 interleaving, i.e., the modules will be partitioned into four groups with three modules in each group. We cannot use 3-3-3 interleaving for 9-module jobs since we need a power of two groups in order to use the last few bits as the group bits. (Recall, however, Table 21 which indicates that for our job mix, very few jobs require more than six modules.)

If we also implement a QR_5 circuit in the processor, we can improve the situation stated above. Especially, the 5-module jobs will no longer need an extra module in order to use the 3-3 interleaving. In Table 29, we show the result with the addition of a QR_5 circuit.

After the generation of a physical address, the processor will send it to all the memory modules attached to its processor bus. Inside each module, there should be some identifying hardware so that the destination module will pick up the address and other modules will reject the address. This can be easily done by using a comparator and an identity register which contains the module number.

Job Size	Number of Modules Required	Interleaving	g	ℓ	Memory Waste
1	1	1	0	0	No
2	2	2	0	1	No
3	3	3	0	0	No
4	4	4	0	2	No
5	6	3-3	1	0	Yes
6	6	3-3	1	0	No
7	8	8	0	3	Yes
8	8	8	0	3	No
9	12	3-3-3-3	2	0	Yes
10	12	3-3-3-3	2	0	Yes
11	12	3-3-3-3	2	0	Yes
12	12	3-3-3-3	2	0	No

Table 28. The Number of Modules Required and Interleaving Scheme for each Job Size. (with only a QR_3 Circuit)

Job Size	Number of Modules Required	Interleaving	g	ℓ	Memory Waste
1	1	1	0	0	No
2	2	2	0	1	No
3	3	3	0	0	No
4	4	4	0	2	No
5	5	5	0	0	No
6	6	3-3	1	0	No
7	8	8	0	3	Yes
8	8	8	0	3	No
9	10	5-5	1	0	Yes
10	10	5-5	1	0	No
11	12	3-3-3-3	2	0	Yes
12	12	3-3-3-3	2	0	No

Table 29. The Number of Modules Required and Interleaving Scheme for each Job Size. (with both QR_3 and QR_5 Circuits)

4.2.2 I/O Connection

The other problem we need to discuss is the connection between processors and I/O devices. Recall in Figure 1, the PRIME system uses an External Access Network to provide the communication paths between processors and external devices. The network is essentially a cross-bar switch, except it also allows two processors to set up a path between themselves. This is done by connecting both processors to a free switch node, i.e., a node that does not connect to any external device [15].

This network is easier to control and allows simultaneous use of all the I/O devices. The probability of access conflict, i.e., more than one processor accessing the same device, will be small if the number of I/O devices is large enough. For example, in the systems we are simulating, the results show that the average number of jobs in the I/O stage is less than the number of I/O devices. This implies that in general a job does not need to wait for an I/O device if we use a network like the EAN to interconnect the processors and I/O devices.

However, the cost of this kind of network will be very expensive if the system size is large. This is the typical disadvantage of a cross-bar-like network. In our simulation model, we choose a common bus structure which is shown in Figure 45. Each processor in this structure is connected to a common bus. The number of common buses we should use is determined by the traffic between processors and I/O devices. Usually, this is proportional to the system size. The I/O devices are partitioned into groups and the I/O devices in one group are connected to an I/O bus. These I/O buses are interconnected with the common buses via a small crossbar switch. The switch allows a processor to access any I/O device.

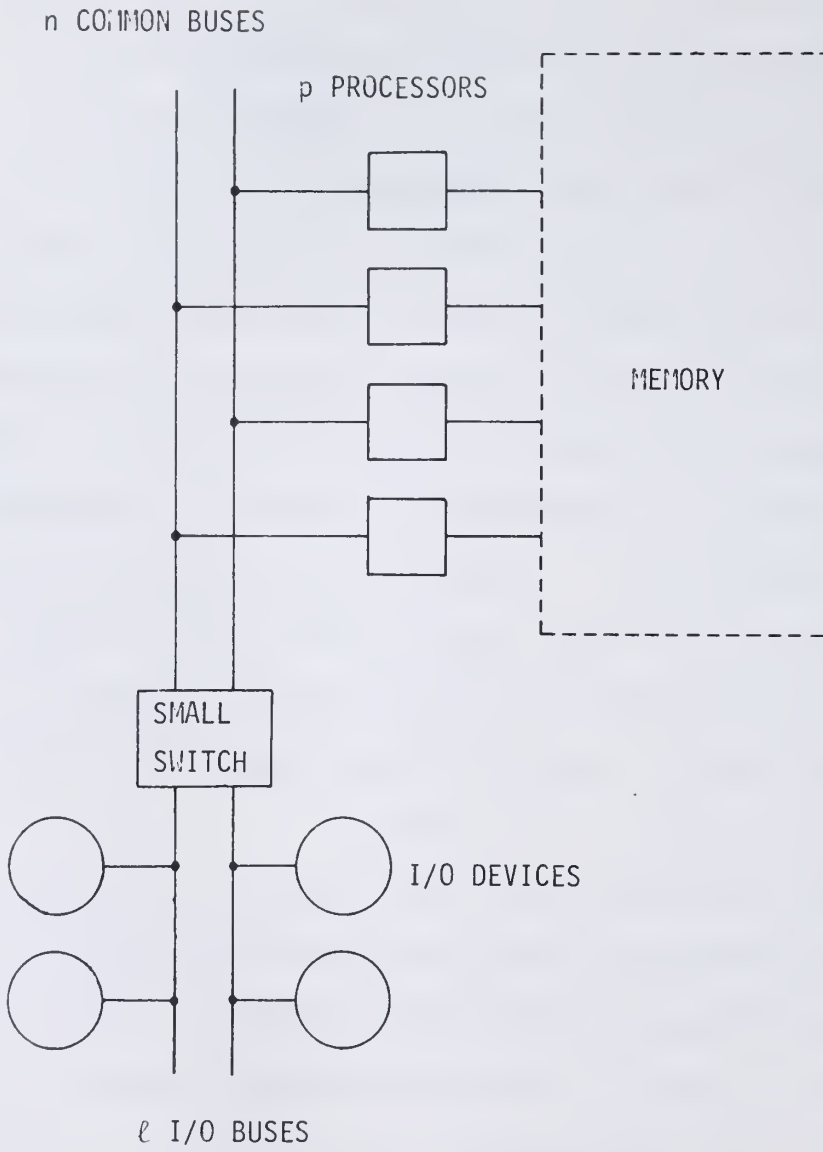


Figure 45. The Interconnection Network between Processors and I/O Devices.

The cost of this design apparently is much cheaper than the cost of a connection network like the EAN in the PRIME system. If we use n common buses and ℓ I/O buses, the total cost will be the cost of $n+\ell$ buses plus the cost of an n by ℓ switch. Suppose n and ℓ are small, this cost will be very low. In addition, when we increase the number of processors, we can simply connect an additional processor to a common bus if the I/O traffic does not increase too much. In an EAN-like network, the extra processor will cause a significant increase of the network size.

Although the sharing of a common bus by a number of processors can reduce the number of buses we need and the size of the switch, bus contention might occur if more than one processor connected to the same bus want to access I/O devices at the same time. The contention can be serious if the average number of I/O requests for a job is large or too many processors are connected to one bus. On the other hand, bus contention might also occur on an I/O bus if more than one processor is trying to access the I/O devices connected to the same I/O bus. Both of these bus contentions will result in queueing of an I/O request, which will cause some delay to a job. Of course, we can keep the bus contentions small by making n and ℓ large enough. In order to find out the number of buses we should use, let us analyze how n and ℓ affect the bus contentions occurring in our connection network.

We will derive the probability that a processor can successfully access the I/O device it wants without being blocked due to bus connections. Let us assume α to be the probability that a certain processor is performing an I/O operation. Roughly speaking, the ratio of the I/O time and the total service time can be thought of as α . Therefore, a processor-bound job mix has a small value of α and an I/O-bound job mix has a large value. We will

assume a to be the same for all active jobs, or processors. Of course, $1-a$ is the probability that a processor will not issue an I/O request. We also assume each common bus has p/n processors attached to it.

Obviously, if a processor wants to make a successful access, the required common and I/O buses should both be free. So, the probability of a successful access is the product of the probabilities of these two events. The probability of the first event is $(1-a)^{p/n-1}$, which is the probability that all other $p/n - 1$ processors sharing the same common bus are not doing I/O. The probability of the second event is

$$\sum_{i=0}^{n-1} \binom{n-1}{i} [1 - (1-a)^{p/n}]^i [(1-a)^{p/n}]^{n-i-1} (1 - 1/\ell),$$

where each term in the summation is the probability that exactly i of the remaining $n-1$ common buses are busy but none of these i requests are accessing the I/O bus we are interested in. This summation is actually the expansion of the following equation:

$$[(1-a)^{p/n} + (1 - 1/\ell)(1 - (1-a)^{p/n})]^{n-1} = [1 - 1/\ell + 1/\ell(1-a)^{p/n}]^{n-1}$$

Hence, the probability of successful access P_s can be expressed as follows:

$$P_s = (1-a)^{p/n-1} [1 - 1/\ell + 1/\ell(1-a)^{p/n}]^{n-1}.$$

In Table 30, we show some numerical values of P_s for different a , n , and ℓ values. Here we use eight processors. As we can see, when a is small, P_s is rather large even for moderate values of n . If we use the definition stated above, we can see the a value for our job mix is about 0.3 since our simulation result shows that the I/O time for a job is roughly 30% of the total service time. If we look in the 0.3 column, we need four

$p = 8$

$n \backslash a$	0.5	0.4	0.3	0.2	0.1
2	0.07	0.12	0.21	0.36	0.60
3	0.08	0.13	0.22	0.37	0.61
4	0.12	0.19	0.29	0.44	0.67
8	0.13	0.21	0.32	0.48	0.70

($\ell=2$)

$n \backslash a$	0.5	0.4	0.3	0.2	0.1
2	0.09	0.15	0.26	0.41	0.65
3	0.13	0.20	0.30	0.45	0.67
4	0.21	0.29	0.40	0.55	0.74
8	0.28	0.37	0.48	0.62	0.79

($\ell=3$)

$n \backslash a$	0.5	0.4	0.3	0.2	0.1
2	0.10	0.17	0.28	0.44	0.67
3	0.15	0.23	0.34	0.50	0.70
4	0.27	0.36	0.47	0.60	0.78
8	0.39	0.48	0.58	0.70	0.84

($\ell=4$)

Table 30. The Probability of Successful Access to
I/O Device.

common buses and four I/O buses in order to obtain a near 0.5 probability. Even with eight common buses, we can only achieve near 0.6 probability. It seems that this design is not very attractive due to the high blocking probability.

However, when we are doing an I/O operation, we do not have to occupy the buses all the time. We can release the buses during the seek and rotational latency time of an I/O operation and let some other processors use the buses. In other words, a processor will only occupy the common and I/O buses when the data or address is being transferred. This effectively reduces the value of α and hence increases the probability of successful access. For example, if the data transfer time is only one-third of an I/O transaction time, α can be reduced from 0.3 to 0.1 and we can get very high probability of a successful I/O access. Of course, we need to implement a smart controller to control the use of these buses.

In fact, if we assume a bus will only be occupied during data transfer, our model of Figure 45 is almost equivalent to the L-M memory organization model proposed by Briggs [50]. The analytic result of his model can be slightly modified to fit our model or used as an approximation of our model.

In our simulations, we used p common buses and two I/O buses, i.e., $n=p$ and $\ell=2$. From all simulation results, we can see the queueing time for I/O devices is relatively low. Apparently, the structure of Figure 45. is quite good. If we use faster I/O devices, we can also reduce the α value. Hence, we can use fewer buses and maintain the same performance.

4.3 Further Problems

Multiprocessor have been an important subject in computer design for quite a while. Recently, new technology has permitted us to consider systems with large numbers of autonomous processors. Most of the work done previously in this area has concentrated on speeding up single program through the use of multiple processors, or on providing a multiprogramming environment. This in turn has led to complex memory-processor and interprocessor communication schemes. We have shown in this thesis that multiprogramming is not necessary for high throughput, low turnaround time, and that simpler architecture is indeed a viable design alternative, producing good performance and expandability, and capable of high reliability and availability.

However, there are still a number of areas which need further study. We discussed the design of several components of a system (e.g., addressing hardware), but many of the details of the processor, memory, and I/O systems need more work. Some of this design is straightforward, but some requires better model before we fully understand the tradeoffs involved. For example, in determining the actual distribution and connection of memories to processors, we have been unable to demonstrate a model which correctly predicts the best distribution or connection. But we have shown that the distribution and connection does cause significant changes in performance.

In this thesis, we have purposely omitted consideration of interactive job loads and virtual memory. An interactive environment imposes new problems both in connecting terminals to the system and in handling the large number of small tasks. This type of environment should be investigated to determine whether or not it would necessitate significant changes to the architecture or our

conclusions.

Finally, a very important research area concerns reliable operating systems. Conceptually, it is easier to design an operating system with centralized control. But this approach leaves us open to total system failure if the control hardware fails. In Chapter 1, we briefly describe the design philosophies used by the PRIME, C.mmp, and NonStop systems. The C.mmp and NonStop systems essentially let each subsystem own a copy of the operating system. This prevents the failure of the operating system of one subsystem from affecting the operations of the other subsystems. However, this duplication does occupy a significant amount of memory. The PRIME system, on the other hand, partitions the operating system into a central control monitor and external control monitors (ECM's) and distributes these ECM's to different subsystems. This distributed approach, of course, can reduce the memory requirement. However, it does create some problems when the control subsystem (the one who is running the central control monitor) goes down, for example, how to save all the system tables and pass control to the subsystem that is taking over the central control monitor. Complete distribution of the central control, or minimization of the central control so it could reside in a highly reliable component, is an interesting and important research area.

References

- [1] Unger, S. H. "A Computer Oriented toward Spatial Problems," Proceedings of the IRE, Vol. 46, No. 10, pp. 1744-1750, October 1958.
- [2] Leiner, A. L., W. A. Notz, J. L. Smith, and A. Weinberger. "PILOT, A New Multiple Computer System," Journal of the ACM, Vol. 6, No. 3, pp. 313-335, 1959.
- [3] "Vocabulary for Information Processing," American National Standard X3, December 1970.
- [4] Enslow, P. H., Jr. Multiprocessors and Parallel Processing, John Wiley and Sons, Inc., New York, 1974.
- [5] Enslow, P. H., Jr. "Multiprocessor Organization - A Survey," ACM Computing Surveys, Vol. 9, No. 1, pp. 103-129, March, 1977.
- [6] Anderson, J. P., S. A. Hoffman, J. Shifman, and R. J. Williams. "D825- A Multiple Computer System for Command and Control," AFIPS Conference of 1962 FJCC, Vol. 22, pp. 86-96, 1962.
- [7] Bell, C. G. and A. Newell. Computer Structures: Readings and Examples, McGraw Hill, New York, 1971.
- [8] Thurber, K. J. and L. D. Wald. "Associative and Parallel Processors," ACM Computing Surveys, Vol. 7, No. 4, pp. 215-255, December 1975.
- [9] Barnes, G. H., R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. E. Stokes. "The ILLIAC IV Computer," IEEE Transactions on Computers, C-17, pp. 746-757, August 1968.
- [10] Slotnick, D. L., W. C. Borck, and R. C. McReynolds, "The SOLOMON Computer," AFIPS Conference Proceedings of 1962 FJCC, Vol. 22, pp. 97-107, 1962.
- [11] Fung, L. "A Massively Parallel Processing Computer," Proceedings of the

Symposium on High Speed Computer and Algorithm Organization, Department of Computer Science, University of Illinois, April 1977.

- [12] Baskin, H. B., B. R. Borgerson, and R. Roberts. "PRIME - A Modular Architecture for Terminal-Oriented Systems," AFIPS Conference Proceedings of 1972 SJCC, Vol. 40, pp. 431-437, May 1972.
- [13] Wulf, W. A. and C. G. Bell. "C.mmp - A Multi-Mini-Processor," AFIPS Conference Proceedings of 1972 FJCC, Vol. 41, Part II, pp. 765-777, 1972.
- [14] "Seven Tough Problems in On-Line Data Base Systems and How Tandem's NonStop System Solves Them," Datamation, 1977.
- [15] Quatse, J. T., P. Gaulene, and D. Dodge. "The External Access Network of a Modular Computer System," AFIPS Conference Proceedings of 1972 SJCC, Vol. 40, pp. 783-790, May 1972.
- [16] Ferrari, D. "Architecture and Instrumentation in a Modular Interactive System," IEEE Computer, pp. 25-29, November 1973.
- [17] Fabry, R. S. "Dynamic Verification of Operating System Decisions," Communications of the ACM, Vol. 16, No. 11, pp. 659-668, November 1973.
- [18] Ravi, C. V. "On the Issue of Physical Memory Sharing in the MCS System," Document No. W-37.0/CSRP, Computer Systems Research Project, University of California at Berkeley, July 1971.
- [19] Bell, C. G., et al. "C.mmp - The CMU Multiminiprocessor Computer," Department of Computer Science, Carnegie-Mellon University, August 1971
- [20] Wulf, W. A., E. Cohen, W. Corwind, A. Jones, R. Levin, C. Pierson, and F. Pollack. "HYDRA: The Kernel of a Multiprocessor Operating System," Communications of the ACM, Vol. 17, No. 6, pp. 337-345, June 1975.
- [21] Abate, J., H. Dubner, and S. B. Weinberg. "Queueing Analysis of the

- IBM 2314 Disk Storage Facility," Journal of the ACM, Vol. 15, No. 4, pp. 577-589, October 1968.
- [22] Oney, W. C., "Queueing Analysis of the Scan Policy for Moving-Head Disks," Journal of the ACM, Vol. 22, No. 3, pp. 397-412, July 1975.
- [23] Fuller, S. H. and F. Baskett. "An Analysis of Drum Storage Units," Journal of the ACM, Vol. 22, No. 1, pp. 83-105, January 1975.
- [24] Bhandarkar, D. P. "On the Performance of Magnetic Bubble Memories in Computer Systems," IEEE Transactions on Computers, Vol. C-24, No. 11, November 1975.
- [25] Coffman, E. G., Jr. and P. J. Denning. Operating Systems Theory, Prentice-Hall, Englewood Cliffs, 1973.
- [26] Kleinrock, L. Queueing Systems, Volume 2: Computer Applications, John Wiley and Sons, New York, 1976.
- [27] Gaver, D. P. "Probability Models for Multiprogramming Computer Systems," Journal of the ACM, Vol. 14, pp. 423-438, 1967.
- [28] McKinney, J. M. "A Survey of Analytical Time-Sharing Models," Computing Surveys, Vol. 1, pp. 105-116, 1969.
- [29] Binder, R., N. Abramson, F. F. Kuo, A. Okinaka, and D. Wax. "ALOHA Packet Broadcasting - A Retrospect," AFIPS Conference Proceedings, 1975 NCC, Vol. 44, pp. 203-215, 1975.
- [30] Mallach, E. G. "Job-Mix Modeling and System Analysis of an Aerospace Multiprocessor," IEEE Transactions on Computers, Vol. C-21, No. 5, pp. 446-454, May 1972.
- [31] Avi-Itzhak, B. and D. P. Heyman. "Multiprogramming Computer Systems," Operations Research, Vol. 21, pp. 1212-1230, 1973.

- [32] Konheim, A. G. and M. Reiser. "A Queueing Model with Finite Waiting Room and Blocking," Journal of the ACM, Vol. 23, No. 2, April 1976.
- [33] Brown, R. M., J. C. Browne, and K. M. Chandy. "Memory Management and Response Time," Communications of the ACM, Vol. 20, No. 3, pp. 153-165, March 1977.
- [34] Franta, W. R. "The Mathematical Analysis of the Computer System Models as a Two-Stage Cyclic Queue," Acta Informatica 6, pp. 187-209, 1976.
- [35] Jackson, J. R. "Jobshop-like Queueing Systems," Management Science, Vol. 10, No. 1, pp. 131-142, October 1963.
- [36] Gordon, W. J. and G. F. Newell. "Closed Queueing Systems with Exponential Servers," Operations Research, Vol. 15, pp. 254-265, 1967.
- [37] Browne, J. C., K. M. Chandy, R. M. Brown, T. W. Keller, D. F. Towsley, and C. W. Dissly. "Hierarchical Techniques for the Development of Realistic Models of Complex Computer Systems," Proceedings of IEEE, Vol. 63, No. 6, pp. 966-976, June 1975.
- [38] Ravi, C. V. "On the Bandwidth and Interference in Interleaved Memory Systems," IEEE Transactions on Computers, Vol. C-21, pp. 899-901, August 1972.
- [39] Chang, D. Y. "Analysis and Design of Interleaved Memory Systems," M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. 75-747, August 1975.
- [40] Hellerman, H. Digital Computer System Principles, McGraw-Hill, New York, pp. 228-229, 1967.
- [41] Burnett, G. J. and E. G. Coffman, Jr. "A Combinatorial Problem Related to Interleaved Memory Systems," Journal of the ACM, Vol. 20, No. 1, pp. 39-45, January 1973.
- [42] Chang, D. Y., D. J. Kuck, and D. H. Lawrie. "On the Effective Bandwidth

- of Parallel Memories, IEEE Transactions on Computers, Vol. C-26, No. 5, pp. 480-490, May 1977.
- [43] Kuck, D. J. The Structure of Computers and Computations, Volume 1, John Wiley and Sons, New York, 1977.
- [44] Knuth, D. E. The Art of Computer Programming, Volume II: Seminumerical Algorithms, Addison-Wesley, Reading, Massachusetts, 1969.
- [45] Flynn, M. J. "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol. C-21, pp. 948-960, September, 1972.
- [46] Sastry, K. V. and R. Y. Kain. "On the Performance of Certain Multiprocessor Computer Organization," IEEE Transactions on Computers, Vol. C-24, No. 11, pp. 1066-1073, November 1975.
- [47] Bhandarkar, D. P. "Analysis of Memory Interference in Multiprocessors," IEEE Transactions on Computers, Vol. C-24, No. 9, pp. 897-908, September 1975.
- [48] Baskett, F. and A. J. Smith. "Interference in Multiprocessor Computer Systems with Interleaved Memory," Communications of the ACM, Vol. 19, No. 6, pp. 327-334, June 1976.
- [49] Gajski, D. D. and C. Vora. "High Speed Modulo 3 Generator," Submitted for publication.
- [50] Briggs, F. A. "Memory Organizations and Their Effectiveness for Multiprocessing Computers," Ph.D. Thesis, Report No. R-768, UILU-ENG 77-2215, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, May 1977.

Appendix A

Assuming we have p processors referencing m memory modules numbered from 0 to $m-1$. Each processor generates s references in every memory cycle. Let a be the probability that the next reference will access the next module in sequence, i.e., $a = \Pr\{r_{i+1} = (r_i + 1) \bmod m\}$, and any other module with the probability $(1-a)/(m-1)$. We will call the former case a "sequential transition" and the later a "nonsequential transition." Also, let $p_{ij}^{(\ell)}$ be the probability that the i^{th} processor will generate a k in the ℓ^{th} position and no j occurs in the first $\ell-1$ positions, i.e., the probability of the event shown in Figure 14.

In the first position, assuming we know all the reference probability p_{ij} , we have :

$$p_{i1}^{(1)} = p_{i1},$$

$$p_{i2}^{(1)} = p_{i2},$$

$$\vdots$$

$$p_{ij}^{(1)} = p_{ij},$$

$$\vdots$$

$$p_{im}^{(1)} = p_{im}.$$

The probability of no j shows up, trivially, is $1 - p_{ij}^{(1)}$. In the second position, consider two kinds of transition and the definition of $p_{ik}^{(2)}$, we will get

$$\begin{aligned}
p_{i1}^{(2)} &= (1 - p_{ij}^{(1)} - p_{im}^{(1)}) \frac{1-a}{m-1} + p_{im}^{(1)} a, \\
&\vdots \\
p_{i(j-1)}^{(2)} &= (1 - p_{ij}^{(1)} - p_{i(j-2)}^{(1)}) \frac{1-a}{m-1} + p_{i(j-2)}^{(1)} a, \\
p_{i(j+1)}^{(2)} &= (1 - p_{ij}^{(1)}) \frac{1-a}{m-1}, \\
&\vdots \\
p_{im}^{(2)} &= (1 - p_{ij}^{(1)} - p_{i(m-1)}^{(1)}) \frac{1-a}{m-1} + p_{i(m-1)}^{(1)} a.
\end{aligned}$$

The first term is the probability of nonsequential transition and the second term is the probability of sequential transition. The probability of no j in the first two positions is:

$$1 - p_{ij}^{(2)} = \sum_{\substack{k=1 \\ k \neq j}}^m p_{ik}^{(2)} = (1 - p_{ij}^{(1)}) \left[1 - \left(\frac{p_{i(j-1)}^{(1)}}{1-p_{ij}^{(1)}} a + \left(1 - \frac{p_{i(j-1)}^{(1)}}{1-p_{ij}^{(1)}} \right) \frac{1-a}{m-1} \right) \right]$$

Actually, we can extend the above result to s^{th} position (if we accept recursive solution) just by replacing 2 by s , we thus have:

$$\begin{aligned}
p_{i1}^{(s)} &= (1 - p_{ij}^{(s-1)} - p_{im}^{(s-1)}) \frac{1-a}{m-1} + p_{im}^{(s-1)} a \\
&\vdots \\
p_{i(j-1)}^{(s)} &= (1 - p_{ij}^{(s-1)} - p_{i(j-2)}^{(s-1)}) \frac{1-a}{m-1} + p_{i(j-2)}^{(s-1)} a \\
p_{i(j+1)}^{(s)} &= (1 - p_{ij}^{(s-1)}) \frac{1-a}{m-1} \\
&\vdots \\
p_{im}^{(s)} &= (1 - p_{ij}^{(s-1)} - p_{i(m-1)}^{(s-1)}) \frac{1-a}{m-1} + p_{i(m-1)}^{(s-1)} a
\end{aligned}$$

All these can be solved recursively, and the probability that i^{th} processor does not reference j^{th} module will be:

$$1 - p_{ij}^{(s)} = \sum_{\substack{k=1 \\ k \neq j}}^m p_{ik}^{(s)} = (1 - p_{ij}^{(s-1)}) \left[1 - \frac{p_{i(j-1)}^{(s-1)}}{1 - p_{ij}^{(s-1)}} a + \left(1 - \frac{p_{i(j-1)}^{(s-1)}}{1 - p_{ij}^{(s-1)}} \right) \frac{1-a}{m-1} \right].$$

If we call this $q_{ij}^{(s)}$, then by the same argument, the bandwidth equation for $s > 1$ will be

$$B_w = m - \sum_{j=1}^m \prod_{i=1}^p q_{ij}^{(s)}.$$

We can solve the above combinatorial problem by using another method, namely, the Markov chain analysis. Figure 46 shows the Markov chain for this problem. Since we are only interested in finding out the probability that no j will show up in $s-1$ transitions, we simply make state j transfer back to itself with probability 1. Hence we have an absorbing Markov chain.

The transition matrix of this Markov chain is given by:

$$T = \begin{pmatrix} \beta & a & \beta & \dots & \beta \\ \beta & \beta & a & \beta & \dots & \beta \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & 0 & \dots & 0 \\ \beta & \beta & \dots & \beta & a & \dots & \beta \\ \vdots & \vdots & & & \ddots & \ddots & \vdots \\ \beta & \vdots & \dots & \dots & \beta & a \\ a & \beta & \dots & \dots & \beta \end{pmatrix}.$$

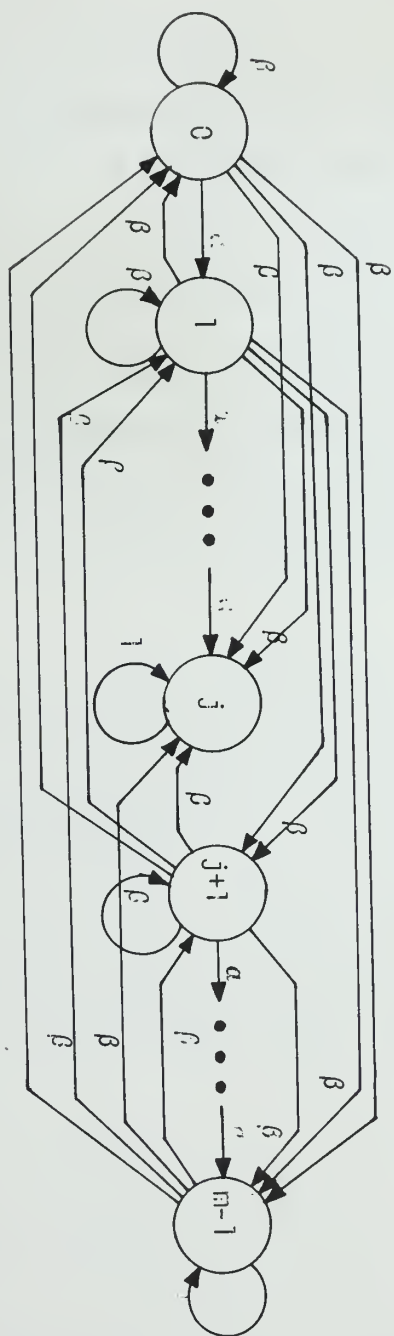


Figure 46. The Absorbing Markov Chain.

If we let π_{ik}^ℓ be the probability that the Markov chain will be in state k after $\ell-1$ transitions, then we can find $\pi_{im}^{\ell+1}$ from the following relation:

$$\pi^{\ell+1} = \pi^\ell T$$

where $\pi^\ell = [\pi_{i1}^\ell, \pi_{i2}^\ell, \dots, \pi_{im}^\ell]$ is the state probability vector.

If we consider the request generation to be the state transition in our Markov chain, then the probability that no j will show up after generating s requests will be equal to $1 - \pi_{ij}^s$, given $\pi^1 = [p_{i1}, p_{i2}, \dots, p_{im}]$. Actually, this method is exactly the same as the previous one, except we are using matrix expression instead of using recursive expression now. Obviously, the latter method is neater and hence more preferable.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-77-908	2.	3. Recipient's Accession No.
	4. Title and Subtitle Further Results Regarding Multiprocessor Systems		5. Report Date October, 1977
7. Author(s) Donald Yi-Chung Chang		8. Performing Organization Rept. No. UIUCDCS-R-77-908	
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Department of Computer Science Urbana, Illinois 61801		10. Project/Task/Work Unit No.	
12. Sponsoring Organization Name and Address National Science Foundation Washington, D. C.		11. Contract/Grant No. US NSF MCS76-81686	
		13. Type of Report & Period Covered Doctoral Dissertation	
15. Supplementary Notes		14.	
16. Abstracts The recent developments of inexpensive but powerful LSI microprocessors and extremely high density semiconductor memory chips have led to the design of large computer systems containing a large number of processors and memory modules. Many systems have been built with many processors interconnected with a large number of memories, e.g., the PRIME system, the C.mmp system, and the Tandem NonStop system. All these systems have one common feature, i.e., to increase the system throughput by simultaneous operation of several processors. We give a brief description of these multiprocessor systems in both hardware and software aspects. Each system has a different architecture and its own advantages and disadvantages. They provide us with many valuable examples of system design. However, a lot of questions arise in this area which are yet to be solved. For example, is multiprogramming better than monoprogramming for system operation? Are there better ways to ~~We are now working on the design of a multiprocessor system which is designed to~~ interconnect processors and memories? How does workload affect system performance, etc.? In this thesis, we shall try to answer these questions in order to get a better understanding of multiprocessor system design. Because the systems we study are highly complex, simulation technique is used to collect the data we need to answer our questions. We shall give a detailed description of our simulator and present a lot of simulation results. We also discuss some logic design problems concerning the real system design.			
~~17b. Identifiers/Open-Ended Terms~~			
17. Key Words and Document Analysis. Interleaved memories Multiprocessors Performance of multiprocessors Processor interconnection			
17c. COSATI Field/Group			
18. Availability Statement Release Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 238	
	20. Security Class (This Page) UNCLASSIFIED	22. Price	

MAR 3 1978



UNIVERSITY OF ILLINOIS-URBANA



3 0112 000447992